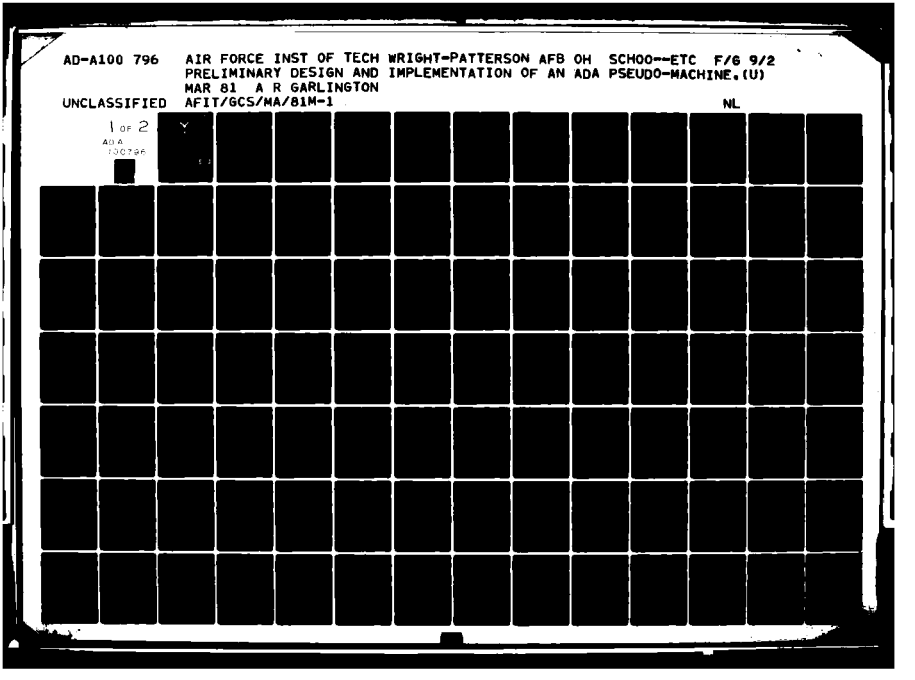

AD-A100 796
AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOO--ETC F/G 9/2
PRELIMINARY DESIGN AND IMPLEMENTATION OF AN ADA PSEUDO-MACHINE.(U)
MAR 81 A R GARLINGTON
UNCLASSIFIED AFIT/GCS/MA/81M-1
NL
1 OF 2
AD A
100796

LEVEL II

UNITED STATES AIR FORCE

AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCS/MA/81M-1

(6)

PRELIMINARY DESIGN AND IMPLEMENTATION OF AN ADA PSEUDO-MACHINE

THESIS

(14) AFIT/GCS/MA/81M-1

(10) Alan R. Garlington

CAPT USAF

(11)

(12) 45

Approved for public release; distribution unlimited.

AFIT/GCS/MA/81M-1

PRELIMINARY DESIGN AND IMPLEMENTATION OF AN ADA PSEUDO-MACHINE

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

in Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Computer Engineering

by

Alan R. Garlington

Capt, USAF

Graduate Computer Science

March 81

Preface

The Department of Defense funded the development of the Ada programming language in an effort to significantly reduce costs for programming embedded computer systems. As the language evolved, it became apparent that its power would be useful in a wide range of other programming applications as well. Since the Department of Defense has a large number and wide variety of small computers, all of which could benefit greatly from Ada, it seemed that a portable compiler could be extremely valuable. One proven implementation of a portable high-level language compiler is UCSD PASCAL which generates code for a hypothetical processor called a pseudo-machine. To run this code on an actual computer requires a software program to simulate the pseudo-machine. This approach allowed PASCAL to be available on many different small computers. Investigation of the same techniques used in this implementation would possibly be the key to providing a portable Ada compiler. Initiating such research at AFIT would provide a test bed for development of this concept and provide a basis for further research in Ada and its programming environment. This seemed like a good prospect for a thesis topic, since both AFIT and the Air Force could benefit.

The preliminary reference manual for Ada was released about the same time I started my first compiler-theory

class. As time passed, my interest in both Ada and compiler theory grew, as did my desire to do a thesis topic that would combine both subjects. When Capt Roie Black and Maj Alan Ross, AFIT professors, voiced interest in the design of an Ada pseudo-machine, I felt that this would be the perfect topic for me. Designing the pseudo-machine would require integrating techniques from several areas in the computer-science field. Compiler theory, data structures and computer architecture are all intertwined, offering a potentially-rewarding, learning experience.

My initial stab at this topic was to design the pseudo-machine and then build a test compiler to exercise the design. This uncoupled development of the pseudo-machine and the compiler turned out to be incorrect.

The process of defining a pseudo-machine for a language is, in fact, closely coupled to the development of a compiler for that language. Niklaus Wirth, designer of PASCAL, stated that the purpose of a pseudo-machine was to "keep the description of the compiler reasonably simple and free from extraneous considerations of peculiar properties of a real, existing processor (Ref 14:331)". If this is the case, defining the Ada pseudo-machine without developing the compiler would be difficult, since the success of the pseudo-machine is measured by how well the machine supports

the compiler.

Thus, the project was modified to permit the parallel development of the compiler with the definition of the pseudo-machine, and the project's scope was reduced. The new goal became an integer-only implementation which incorporated some of Ada's more interesting features, for example, packages and tasks. Additional constructs were studied based on their estimated impact on the pseudo-machine's architecture.

Throughout the project's development, several people sacrificed their time to offer council, suggestions, and materials. The time they invested immeasurably improved the project's overall quality. Thanks to Capt Roie R. Black, advisor, who proposed the topic and helped limit its scope. His interest, encouragement and suggestions were invaluable. Thanks to Major Dan Burton, currently assigned to the Air Force Avionics Lab, who sponsored the project and allowed me to use the lab's excellent computer facilities. Thanks also to my thesis-committee members, Lt Col Jim Rutledge and Maj Alan Ross. Lt Col Rutledge's indepth knowledge of Ada proved very valuable. He sacrificed numerous hours listening and teaching, and he also composed several test programs to help debug the project. Major Ross served as a thoughtful critic who, in concert with Capt Black, identified the Ada

pseudo-machine as a possible thesis topic. Finally, I'd like to thank my wife whose patience with a full-time student and a baby boy (born during my first quarter at AFIT), was nothing short of miraculous.

Table of Contents

1. Introduction 1

1.1 Background -- Ada's roots 1
1.1.1 The problem that spawned a language 1
1.1.2 A solution in commonality 2
1.1.3 Ada's broadening appeal 3
1.1.4 Ada and the small computer 7
1.1.5 Needed -- a portable compiler 8
1.1.5.1 The pseudo-code compiler 9
1.1.5.2 Advantages/disadvantages 10
1.2 Project overview 11

2. Project Description 13

2.1 Approach 13
2.2 Design considerations 14
2.2.1 Recursive subprograms 14
2.2.2 Tasking 16
2.2.2.1 Ada's tasking facility 17
2.2.2.2 Run-time requirements 21
2.3 Pseudo-machine architecture 23
2.3.1 Memory 28
2.3.1.1 Program memory 28
2.3.1.2 Stack memory 29
2.3.2 Stack processors 32
2.4 Pseudo-machine instruction set 35
2.4.1 Relational operators 36
2.4.2 Integer (single word) arithmetic operations 37
2.4.2.1 Single word loads and stores 37
2.4.2.2 Arithmetic operators 38
2.4.3 Tasking operators 39
2.4.3.1 ACTIVATE 39
2.4.3.2 CALLENTRY 41
2.4.3.3 ACCEPT 42
2.4.3.4 RELEASE 42
2.4.3.5 TERMINATE 43
2.4.3.6 ENTILOAD 43
2.4.3.7 ENTISTORE 44
2.4.4 I/O Operations 44
2.4.4.1 SPUT 44
2.4.4.2 IPUT 45
2.4.4.3 IGET 45
2.4.5 Miscellaneous Instructions 45
2.4.5.1 CALL 45
2.4.5.2 PARAMSHIFT 46
2.4.5.3 RETURN 46

TABLE OF CONTENTS

2.4.5.4 JMP 47
2.4.5.5 JMPF, JMPT 47
2.4.5.6 INCT 47
2.5 The compiler 48
2.5.1 Background -- Compilation 48
2.5.2 LR(1) parsing automaton 52
2.5.2.1 Construction 53
2.5.2.2 Parser structure 54
2.5.2.3 Parser operation 55
2.5.3 Semantic routines 56
2.5.3.1 Scanner 56
2.5.3.2 Semantic stacks 57
2.5.3.3 Sample semantic routine 57
2.5.4 Symbol table and visibility 59
2.5.4.1 Environment stack 59
2.5.4.2 Stacking rules 59
2.5.4.3 Symbol-table routines 61
2.5.4.4 Visibility example 62

3. Recommendations 65

3.1 Improvements to the pseudo-machine 65
3.1.1 Run-time space allocation 65
3.1.2 System queues 66
3.1.3 Stack-frame control data 66
3.1.4 Implementing exceptions 66
3.1.5 Implementing dynamic variables 67
3.1.6 Enumeration I/O 67
3.1.7 Data protection 68
3.2 Improvements to the compiler 68
3.2.1 Towards a finished product 68
3.2.2 For use as a tool 70

BIBLIOGRAPHY 72

1 APPENDICES 73

I. DOD Commonality study 74

II. LR(1) Parsing automaton 77

III. User's guide 85

IV. Source listing 91

VITA 92

List of Figures

Figure 2-1:	Run-Time Space Allocation Example	15
Figure 2-2:	Procedure Activation Record	16
Figure 2-3:	Task Declaration	17
Figure 2-4:	Accept Statement	18
Figure 2-5:	Parent With Nested Task	21
Figure 2-6:	System Architecture	23
Figure 2-7:	Correspondence of Registers and Blocks	26
Figure 2-8:	Sample Task Frame	33
Figure 2-9:	Truth Table for the Binary, Relational Operators	36
Figure 2-10:	Truth Table for the Unary-Operator ZNOT	36
Figure 2-11:	Parser-Structure Chart	54
Figure 2-12:	Algorithm for Module Parse	55
Figure 2-13:	Stacking Rules	60
Figure 2-14:	Example Program for Visibility Demonstration	62
Figure 2-15:	Visibility Rules Demonstration	63
Figure 3-1:	Tables for the LR(1) Parsing Automaton	80
Figure 3-2:	Result of the 'Shift, T1' Move	81
Figure 3-3:	Intermediate result of the 'Reduce, 3' Move	82
Figure 3-4:	Final result of 'Reduce,3' Move	82
Figure 3-5:	Acceptance of the String "JOHN EATS BREAD"	83

ABSTRACT

Abstract

This project involved defining an Ada pseudo-machine and developing an Ada to pseudo-code test translator. The translator's front-end incorporates a table-driven parser that can parse the entire proposed-standard Ada language. The translator's semantic routines allow integer data objects, several control structures, procedures, functions, packages and tasks. These routines generate pseudo-code that is executed by an interpreter program included in the translator. The interpreter constitutes a complete description of the pseudo-machine whose architecture consists of multiple, stack-oriented processors that access a common memory. Interesting features of the project include the hash-coded symbol table that supports Ada's visibility rules and the pseudo-machine architecture that supports Ada's tasking.

INTRODUCTION

1. Introduction

Chapter one begins with background information on the origin of the computer programming language Ada and with a summary of the features which have resulted in its rapid growth in popularity. The chapter concludes with a description of how Ada can be implemented on a micro or mini computer and with an introductory description of the thesis project.

1.1 Background -- Ada's roots

Increasing software costs have forced the Department of Defense to search for budget reducing strategies. An outgrowth of this search was the development of a new computer language with a soon to be familiar name, Ada. Its introduction met with both opposition and enthusiasm, but currently, most of the opposition is fading and Ada is experiencing a broadening appeal.

1.1.1 The problem that spawned a language

Every year, more and more money is spent on computer systems, despite precipitous drops in computer hardware costs. These falling hardware costs have been matched by an equivalent increase in software costs, and now, software looms at the forefront as the major development expense. The Department of Defense is not immune to this problem, and it too must face the problem of rising expenditures.

Within DOD, the major software expense is for programming embedded computer systems, and more than 50% of the software budget goes toward satisfying the demand for such software (Ref 4:6). Why is programming embedded systems so costly? A preliminary study showed that one of the problems has been the plethora of special-purpose languages and systems used to program them. The study showed that a common language could save in excess of 1 billion dollars a year (Ref 5). The data on which this estimate is based can be found in appendix I.

1.1.2 A solution in commonality

To correct this problem, DOD pursued the goal of a single, high-level computer language appropriate for programming embedded systems and established the High Order Language Commonality Program in 1975 to railroad the project (Ref 3:i). The program's directors compiled a set of requirements for the proposed language by circulating a request for language requirements throughout the military, civilian and industrial communities of the U.S. and, also, throughout European and NATO countries. The submitted requirements were integrated, refined and then returned to the respondents for approval. After several such cycles, the resulting set of requirements was deemed necessary and sufficient for all DOD embedded computer applications.

Several existing high-order languages were examined to see if one satisfied the complete set of requirements, but none did. Therefore, four contractors were funded to develop a language which would meet the required specifications. After this four-contractor competitive-design effort, a single language emerged and was named Ada. Currently, Ada stands at a major transition point in its development, as emphasis shifts from the design of the language to its introduction and use.

1.1.3 Ada's broadening appeal

The major goal of Ada's design was to reduce costs for embedded-computer software. While pursuing this goal, DOD specified requirements for a working environment for the language. These requirements are delineated in the document: "STONEMAN, Requirements for Ada Programming Support Environments", Feb 1980. The stated purpose of the programming-support environment is to "support the development and maintenance of Ada applications software throughout its life cycle, with particular emphasis on software for embedded applications" (Ref 3:1). To meet this goal, a host/target approach to software construction is adopted. This approach entails developing programs for an embedded target computer on a host computer that offers extensive support facilities (Ref 3:8). The document goes on to specify what support facilities the host system must

have.

While this standard environment will do much to make Ada a powerful and popular tool for programming embedded systems, the language has features that have opened another, perhaps-larger, market. This market is among system and applications programmers who develop software for use on the host processor.

Ada has several features that have stimulated this interest. Some of the most interesting of these will be discussed briefly. These include packages, tasks, separate compilation and the promise of universality. More detailed information can be found in the Ada reference manual (Ref 2).

Packages: Packages are a mechanism for isolating logically related structures in the program text whether they are subprograms, data types, variables, nested packages or a combination of these. This mechanism clearly distinguishes information that is accessible only within the package from that which is accessible to the rest of the program. Information accessible only within the package is considered 'hidden' and cannot be altered or even accessed outside the package. This provides the programmer with a powerful tool for creating abstract data types whose implementation details are completely hidden from the user,

for writing modular programs with enforced module boundaries and for writing programs that can be more easily verified (Refs 2:7-1 - 7-11; 9:8-1).

Tasks: Tasks are Ada's construct for parallel processing. A task is known to other tasks in the program by a set of names called entries. When one task calls another, one of these names must be specified. Synchronization of the two tasks is achieved when the called task accepts the call of that named entry. This mechanism provides the programmer with a simple, but powerful, method for specifying parallel activities that must communicate (Ref 2:9-1 - 9-16).

Separate compilation: Ada supports separate compilation as opposed to independent compilation. The distinction between the two was first made by J.J. Horning and is described in the preliminary reference manual. The description is reproduced here for convenience.

> Independent compilation has been achieved by most assembly languages and also by languages such as Fortran and PL/1. Compilation of individual modules is performed independently in the sense that such modules have no way of sharing knowledge of properties defined in other modules.
>
> Independent compilation is usually achieved with a lower level of checks between units than is possible within a single compilation unit. In consequence, independent compilation came into disrepute and was rejected by safety minded, early typed language definitions such as Algol 68 and

> Pascal. Fast compilation of the complete program was often advocated by promoters of these languages as a safe alternative to independent compilation. Fast compilation, however, has its limits, and it fails to answer the needs of confidentiality and libraries.
>
> Separate compilation, on the other hand, reconciles type safety and the pragmatic reasons for compiling in parts. It is based on the use of a library file which contains a record of previous compilations of the units which form a program (Ref 9:10-1).

In summary, independent compilation provides little or no checking for compatability between individually compiled modules. On the other hand, separate compilation provides the same level of checking between individually compiled modules as they would get if they were compiled together.

Universality: Ada holds the promise of being a universally accepted standard language. The list of benefits that could be derived from such a language would be very long, and one can only hope that this will indeed come to pass (Ref 4). DOD's mandate that Ada will be "the" language for programming embedded systems is the foot in the door for such a possibility. Hopefully, this door will open completely as DOD and commercial interests cash in on the language's powerful features.

Collectively, these features support the concept of standard software components first espoused by M.D. McIlroy in 1969, and enthusiastically supported by Jean D. Ichbiah

at the ACM SIGPLAN conference on Ada in 1980. Over-simplifying, a software component refers to a functional module that has been coded and verified. Such a component could be "manufactured" by a specialty company and then be compiled with other such components into software catalogs. On the "consumer's" side, software developers could refer to these catalogs and choose those components needed for their project. Once delivered, the components could be "wired" together to create the finished system. Since the functional modules are already assembled, tested and, possibly, guaranteed by the manufacturer, less time should be necessary for system development and testing. Ada's features have sparked the hope that standard components may soon be a reality.

1.1.4 Ada and the small computer

A large, potential market for software components, and thus for Ada, exists among micro and mini computer users. However, these users must have Ada running on their machines in order to use these components. Since small machines generally cannot support an extensive environment, they must have a specially designed Ada language environment tailored to their capabilities.

Hosting Ada on small machines is a topic reminiscent of the UCSD PASCAL effort where PASCAL, a language seemingly

too complex to host on a micro, was quite successfully implemented. The excellent results achieved by the UCSD PASCAL project served to motivate this investigation of the same techniques as applied to Ada. The project reported in this paper specifically deals with the concurrent development of an Ada pseudo-machine and an Ada to pseudo-code translator, with emphasis on the pseudo-machine's architecture. Some of the results of the project, namely the pseudo-machine definition and the semantic routines from the translator, could serve as a good starting point for the implementation of an Ada environment on a micro-computer host. The following section describes the pseudo-machine approach and then introduces the project more fully.

1.1.5 Needed -- a portable compiler

The most important part of this micro/mini Ada environment is the compiler. Since there is a variety of processors in the micro and mini class, the compiler program should be written incorporating techniques that enhance its portability. A technique that achieves this, with excellent results, is the pseudo-code compiler. In this technique, the compiler generates code (pseudo-code) for a hypothetical processor. This processor's instruction set is specifically designed to make the compilation task easier and more straight-forward than attempting to generate code for an

actual processor. The pseudo-code is then executed by an interpreter program which runs on the actual processor. The steps necessary to develop such a compiler are briefly outlined below.

1.1.5.1 The pseudo-code compiler

The first step toward developing a pseudo-code compiler is to write the Ada to pseudo-code compiler program in a suitable language on a host processor. The resulting program, called a cross compiler, accepts Ada source text as input and generates pseudo-code for the hypothetical processor as output. Now, a production-quality compiler is written in Ada. This program is input to the cross compiler and translated to pseudo-code. The output from this step is pseudo-code for the production compiler. Now, all that is needed to execute the compiler program is a pseudo-machine. Since no pseudo-machines actually exist yet, a simulator must be created. This may be done by writing a program that accepts the pseudo-code as input and that accomplishes the necessary actions. Such a program is sometimes called an interpreter. With this approach, all that needs to be done to install the Ada compiler on a new processor is to write a relatively-simple, pseudo-code, interpreter program to run on the processor.

1.1.5.2 Advantages/disadvantages

Since installing the compiler on another processor is limited to writing a relatively-simple program, the goal of easy portability is attained. All system software, e.g. the editor, debugger, etc., should also be written in Ada and compiled to pseudo-code, thus attaining the same portability as the compiler.

Of course, running the interpreter program decreases execution speed as compared to executing native machine code. The trade off is speed for portability. However, the popularity of current PASCAL pseudo-code implementations is a testimony that, for most purposes, this degredation in performance is acceptable to a wide market of users. However, should performance be degraded to unacceptable levels, speed can be improved by post processing the code or, even better, by creating the hypothetical processor.

Post processing: Pseudo-code generated by the compiler can be post processed to produce code for the target machine. An optimizing, pseudo-code to target-machine translator would eliminate the run-time, interpretation penalty; thus increasing the program's execution speed.

Building the hypothetical processor: Building the hypothetical processor eliminates any need for post processing the pseudo-code to speed up execution.

Pseudo-code generated by the compiler is native machine-code for the hypothetical processor, and thus, no translation is required. Western Digital Corporation used this technique to build their PASCAL Micro-Engine by micro-programming an LSI 11 chip set to emulate the UCSD PASCAL pseudo-machine. The excellent result is a premium, high-speed, UCSD-PASCAL implementation that executes on a micro/mini computer.

Now that some background information on pseudo-code compiling has been covered, the thesis project will be described. This project, while not directly involved with building the finished compiler for an Ada pseudo-code implementation, has laid the foundation for such an effort.

1.2 Project overview

The project consisted of the concurrent development of an Ada to pseudo-code translator and an Ada pseudo-machine. First, a parser for the language was built using Lawrence Livermore Laboratory's automatic parser generator and Intermetric Inc.'s LR(1) Ada grammar (Refs 12, 13). Then, language constructs were investigated beginning with packages. Packages had no impact on the runtime architecture of the pseudo-machine, but profoundly affected the compiler's symbol table and symbol-table-access routines. Separate compilation and its possible impact on the structure of the symbol table was not investigated.

Next, expressions were implemented followed by procedures and functions. These features delineated the basic requirements for the pseudo-machine, and a stack processor was selected to meet those requirements. Finally, tasks were investigated. Tasks added several operators to the pseudo-machine's instruction set and prompted a multi-processor architecture. The following chapter describes each of the project's two parts: the pseudo-machine and the compiler.

2. Project Description

This chapter begins by describing the approach to the project and by discussing some of the design considerations that prompted the resulting system architecture. Later sections describe the pseudo-machine's architecture, including its instruction set and describe the Ada to pseudo-code compiler.

2.1 Approach

As mentioned before, the approach taken in this effort was to develop an Ada compiler in parallel with the pseudo-machine. Experience gained through this approach showed that the compiler actually drove the design of the machine. As each Ada construct was implemented in the compiler, a set of run-time actions became necessary to carry out the required semantics of the construct. This set of actions, where feasible, was included as a single instruction in the machine's instruction set. Where this was not practical, combinations of previously defined instructions were used to implement the construct. In this way, the instruction set for the pseudo-machine grew specially tailored to the needs of the Ada language.

2.2 Design considerations

This approach uncovered two constructs that had major influences on the architecture of the machine. These constructs included the requirement for subprograms to be recursive and the run-time requirements imposed by the tasking constructs.

2.2.1 Recursive subprograms

When a compiler analyzes a subprogram in preparation for code generation, the only information available to it concerning local variable and object space requirements is the space required for a single activation of that subprogram. Since subprograms in Ada can be activated recursively, the total number of times a procedure will be called is not known until run time. Since the compiler cannot allocate variable and object space for an unknown number of subprogram activations, a run-time allocation scheme is required. This scheme must allocate memory space for each variable or object declared in a subprogram before executing any of that subprogram's code. The system used in the pseudo-machine is described below.

Ada subprograms, like those in other block-structured languages, obey a last-called, first-completed calling discipline. That is, the most recently called subprogram must complete its execution before the caller resumes its

execution. Since space for variables and objects is allocated just prior to the subprogram's execution and deallocated immediately afterward, a stack can be used. Consider the following example:

```
PROCEDURE MAIN IS
   A : INTEGER;
 BEGIN
   A := 2;
   PUT_LINE (A);
 END MAIN;
```

Figure 2-1: Run-Time Space Allocation Example

In this example, procedure 'MAIN' merely initializes the local variable 'A', prints it and terminates. The run-time actions necessary to accomplish this in the pseudo-machine are:

1. Allocate stack space for the procedure's local variable 'A'.
2. Allocate temporary storage for the constant 2.
3. Store this value in the space allocated to variable 'A'.
4. Print the value of variable 'A'.
5. Deallocate the procedure's stack space.

When stack space is initially allocated to the procedure, as in step one, additional space is also allocated to provide storage for control data used by the run-time system. The number of words required for control

information is fixed and is allocated before the procedure's local variable space. In this paper, the first word allocated to a procedure is referred to as the base of the procedure activation, and the control data is referred to as stack-frame control data. The space allocated to temporaries, as in step 2, is accessed strictly as a stack, and the last word allocated for temporary storage is called the 'top of stack' or 'T' for short. This will be discussed more fully in a later section.

Collectively, the stack-frame control data, the variable and object space, and the temporary workspace, is called a procedure activation record or a stack frame. The layout of the stack frame for a typical procedure activation is illustrated in figure 2-2. Please note that the stack grows downward here and in all illustrations that will follow.

2.2.2 Tasking

Before discussing the run-time requirements levied by Ada's tasking facilities, a brief introduction to them is necessary.

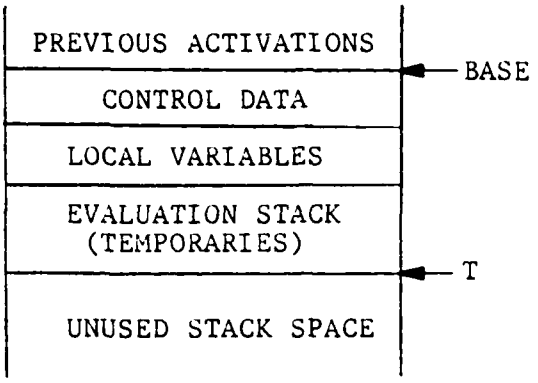


Figure 2-2: Procedure Activation Record

2.2.2.1 Ada's tasking facility

Tasks are the Ada construct used to specify code that executes in parallel with the parent procedure. These parallel tasks may execute totally independently, or they may synchronize occasionally to pass information. The mechanism these tasks use to synchronize and pass data is described in the following paragraphs.

A task that accepts calls from other tasks has a set of pre-defined names that it can be called by. These names are called entries, and their declaration looks the same as a procedure declaration. Figure 2-3 contains the declaration of a task named 'A' that includes 2 such entries.

Task 'A' can be called by other tasks with the entry names 'A.A1' or 'A.A2'. To call task 'A', the caller

```
TASK A IS
  ENTRY A1 (A11 : IN INTEGER);
  ENTRY A2 (A12 : OUT MY_KIND);
END A;
```

Figure 2-3: Task Declaration

executes a call to the desired entry. Actual parameters of the call must match the declaration's parameters in number and type as they must in a procedure call (excluding parameters with default values). This entry call looks identical a procedure call, and might appear as:

```
A.A2 (Z)
```

where Z must be of type 'MY_KIND'. The called task accepts such a call by executing an accept statement for the called entry. Continuing the example, such an accept statement might appear as:

```
ACCEPT A2 (A21 : OUT MY_KIND) DO
      .
      .
  STATEMENTS
      .
      .
  END A2;
```

Figure 2-4: Accept Statement

The region between 'DO' and 'END' in the figure is called the accept body, and, in this region, the two tasks

are synchronized. During this rendezvous, the statements comprising the accept body are executed while the calling task is suspended at the point of call. After the completion of the accept body, both tasks continue their parallel execution. Now that the mechanism for synchronizing parallel tasks has been described, the method for passing data will be discussed.

Information may be passed between communicating tasks in two ways; through entry parameters or through global variables. The preferred way to pass information is by referencing the entry parameters. In this technique, the entry parameter acts like a local variable in the accept body. Entry parameters, like procedure parameters, can have a specified mode which can be 'IN', 'OUT' or 'IN OUT'. Thus, information passes into the called task via an actual parameter corresponding to an 'IN' or 'IN OUT' formal parameter, and passes out of the task via an actual parameter corresponding to an 'OUT' or 'IN OUT' formal parameter. This method is safe since the two tasks have rendezvoused and are in direct communication with each other. The other technique for passing information between tasks is by accessing global variables. There are no guarantees with this technique, and the programmer must implement control constructs and safeguards. Let the user beware!

The chronology of the calling interaction between tasks permits two possible scenarios (Ref 7:1). In the first scenario, the calling-task's entry call precedes the called-task's execution of an accept statement for that entry. The second scenario is the opposite, where the called task executes an accept statement for an entry and then the caller executes the entry call. The required run-time actions for each scenario are briefly discussed below.

Call precedes accept: If the caller executes an entry call and finds the called task unable to accept the call, it enters itself in a wait queue associated with that entry. Then it records the current value of its working registers (its context) in its stack frame and releases its processor. The scheduler then attempts to schedule the released processor.

Accept precedes call: The task owning the entry executes an accept statement and finds there are no callers. It then sets a flag notifying other communicating tasks that it is waiting for a call to that entry, records its context and releases its processor. The scheduler then attempts to schedule the released processor.

This brief introduction to Ada's tasking facility will now be followed by a description of the run-time operations

required to implement these constructs. If more information on tasking is desired, see the Ada reference manual (Ref 2:9-1 - 9-16).

2.2.2.2 Run-time requirements

In the pseudo-machine, an Ada task executes as an object within a subprogram's activation record. As a result of this, the stack space assigned to the parent must be accessible to multiple processors. This is true because one processor might require access to the variable/object space allocated to the task object, while another processor might require access to other areas of the parent's variable/object space.

The stack space allocated to a task is formatted the same as that of a procedure, except that the task has a pre-established upper bound on the amount of space assigned to it. This bound is determined by the compiler when the program is compiled, but can be modified by a representation specification for the task (Ref 2 : 13-3). Now, consider an example showing the stackspace allocated to a task object nested within its parent.

Notice that the nested task in figure 2-5 has the same structure as the parent and is nested within the parent's local variable space. Remember that the space allocated to the nested task has definite limits that are computed by the

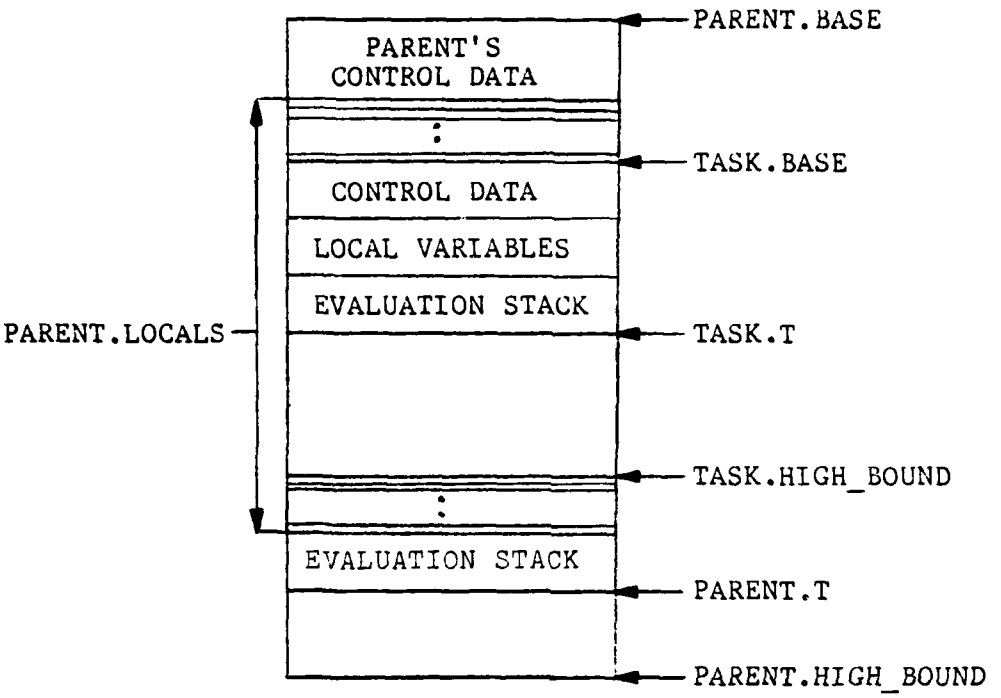


Figure 2-5: Parent With Nested Task

compiler at compile time. If subprograms executing within such a nested task recurse excessively, the space allocated to the task will be exhausted and execution must stop. The program would then have to be recompiled, this time notifying the compiler that the nested task needs more space. A better, more complicated, solution would be to develop a run-time space allocation scheme to cover such occurences. However, space is currently computed at compile time only, and there is no run-time space allocation scheme.

An earlier section on recursive subprograms hinted strongly that the pseudo-machine required a stack-oriented processor, and the previous section on tasking hinted at multiple processors. The following section will describe the resulting combination which comprises the system architecture.

2.3 Pseudo-machine architecture

The pseudo-machine consists of multiple, stack-oriented processors accessing a common, partitioned memory through a controller which resolves conflicts. The following diagram illustrates this architecture.

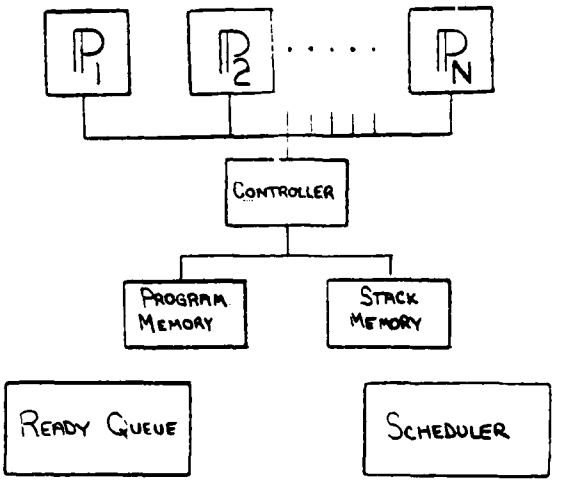


Figure 2-6: System Architecture

The figure illustrates 'n' processors accessing a common, partitioned memory. The stack memory constitutes a

single stack that is shared among the processors, and the program memory contains the code for a single Ada program. The system also has a ready queue for tasks waiting to begin or continue their execution. Associated with the ready queue is a scheduler who assigns waiting tasks to idle processors until exhausting the supply of either tasks or processors. It is implemented as a procedure which is called by some of the pseudo-machine's instructions.

The role of the scheduler can be summarized as follows. When the pseudo-machine begins executing an Ada program, there is only a single thread of control for the main procedure. If that procedure spawns tasks, they are initiated by an ACTIVATE instruction that enters the spawned tasks in the ready queue and calls the scheduler. If these tasks must suspend their execution, for example while waiting for a rendezvous, they release their processor and call the scheduler. The scheduler is also called when a task terminates its execution. Briefly, the scheduler is called when a task is activated, blocks or terminates.

Please note that the structure illustrated on the diagram is logical only. The actual physical structure of the system can be quite different as long as the logical structure is preserved. Each of the 'n' system processors is stack oriented, and their operation will now be

described.

Stack processors get their name from the way they evaluate arithmetic expressions. They evaluate expressions in post-fix, using a stack to store operands and intermediate results. When evaluating such an expression, a stack processor pushes each operand it encounters onto an evaluation stack in its memory and uses each operator it encounters as an instruction. When executing an instruction, it assumes the operands are already on the stack (This is true because of the properties of post-fix evaluation.), and it removes the operands and pushes the result back onto the stack. Since the processor knows where the operands reside, and since it knows where to put the result, the processor requires no addresses to execute such an instruction. For example, the expression 2 * 3 + 9 becomes 2, 3, *, 9, + in post-fix, which translates to PUSH 2, PUSH 3, MULTIPLY, PUSH 9, ADD in stack machine instructions. The result of the expression now resides on top of the stack and is available for assignment to a variable, comparison with other values or for whatever use that can be made of an expression's result.

When a stack machine is used to implement a block structured language, such as Ada, another stack-like feature exists. Since procedure activations obey a last-in,

first-out (LIFO) discipline, stack space is allocated to them in a LIFO manner. These space allocations and deallocations are strictly stack-like, but within the space of each allocation, words are accessed in a controlled but hardly stack-like manner. Values may be removed from the evaluation stack and transferred into the local variable space, or removed from the local variable space and placed on the stack or even stored within another procedure's activation record. Additionally, control data can be accessed or changed at random.

The stack processors in this project contain registers to facilitate access to the data. These registers match closely with the boundaries between the previously described blocks of data contained in a procedure activation. To describe these registers in a specific context, consider the task activation in figure 2-7.

The first word of the activation record is marked by a 'BASE' register and the top of the evaluation stack is marked by a 'T' register. Remember that the evaluation stack grows downward toward the high boundary on the diagram. The task's high boundary is initially marked by the 'HEAP' register. However, the 'HEAP' register's primary purpose is not just to mark the initial high boundary of the task's stack space. Notice that the 'HEAP' register on the

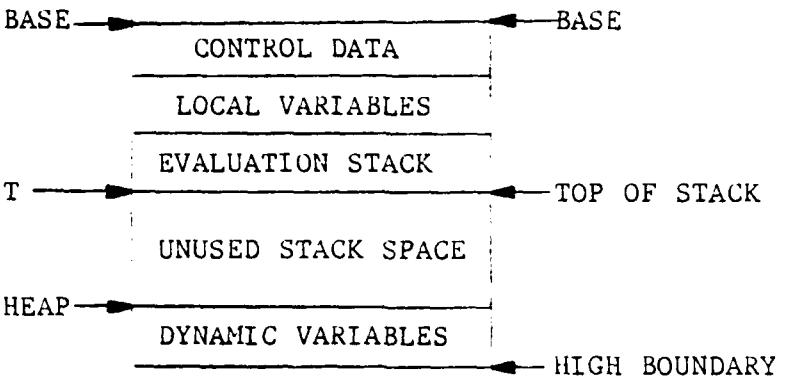


Figure 2-7: Correspondence of Registers and Blocks

illustration has already moved away from the high boundary.

The primary purpose of the 'HEAP' register is to mark the top of the dynamic heap. The dynamic heap provides storage space for program variables created at run time, like those created dynamically by the 'NEW' operator in PASCAL. As these variables are created, space is allocated to them from this heap structure. On the previous diagram, the heap grows upward toward the stack.

Note that the boundary between the control data and the task's local variables/objects is not marked by a special register. The reason for this is that control data is of known length at compile time, and therefore, the offset to the first local variable can be computed.

Now that some background information on stack processors has been covered, more detailed information on the project's pseudo-machine can be presented.

2.3.1 Memory

The memory contains two parts, the stack memory and the program memory. As mentioned previously, any system processor must be able to access any word in stack memory. The following section shows that this is also required of the program memory.

2.3.1.1 Program memory

Program memory contains only the instructions to be executed by the processors. No variable space is allocated within program memory, and only read operations are performed on it. Thus, an actual implementation could hold any program in a read-only memory.

Multi-tasking requires the program memory to be accessible to multiple processors. This is true because Ada's visibility rules permit procedures to be global to multiple tasks, and therefore, two tasks executing in parallel could call a single procedure at the same time. Making duplicate copies of the code is also a possibility, but this is less straight-forward.

2.3.1.2 Stack memory

Stack memory contains space for stack-frame control data, program variables, temporaries, and dynamically allocated variables.

Stack-frame control data: A stack frame is allocated upon entrance to a block, to a subprogram or upon initialization of a nested task object. This stack frame contains the following information:

1. The static link: The static link records the textual nesting level of the program as it was originally written. It is used for run-time addressing of variables and objects.

2. Dynamic link: The dynamic link marks the base of the calling procedure's activation record. It is used to deallocate stack space upon completion of the procedure's execution.

3. Program counter: Storage space is provided for the current value of the processor's working registers. This is necessary since a task may have to give up its processor at any time. For example, a higher priority task may pre-empt this task, forcing it to release the processor. The current values of the processor's working registers must then be stored so that the interrupted task can resume its execution at a later time. The 'program counter' slot is used to store the current value of the 'PC' register should this occur.

4. Task flag: The task flag is a Boolean variable that indicates whether or not the stack frame is a task. It is used to indicate task boundaries when processing run-time exceptions raised in the program.

5. Active nested task counter: The active nested task counter is used to record the number of nested tasks currently active in the given stack

frame. The definition of the Ada language states that a block cannot be exited until all nested tasks have completed their execution (Ref 2 : 9-5). The counter is used to enforce this rule at run time.

6. Waiting flag: The waiting flag is a Boolean variable that indicates whether or not the parent task is waiting to terminate its execution. For example, if the parent reaches the end of its code, and it still has active nested tasks, it must wait for them to complete. It then sets the waiting flag to true, stores the current values of its processor's working registers, and releases its processor.

7. Exceptions: Ada allows controlled error processing at run time through its exception facility. This control word is currently not used since exceptions are not implemented. When exceptions are implemented, the word will be used to record information on exceptions handled within the block. More control words will be required to accomplish this.

8. Priority: This word is a run time record of the task's priority.

9. Top of stack: The top-of-stack control word provides temporary storage for the processor's 'T' register.

10. Base: The base control word provides temporary storage for the processor's 'BASE' register.

11. Link: When a task is entered into a queue, the link control word points to the next task waiting in the queue. A single link field is sufficient since it is a characteristic of the language that a task can be waiting in only one queue (Ref 7 : 11-44).

12. Heap: The heap position provides temporary storage for the processor's 'HEAP' register.

13. Data lock: This Boolean variable indicates whether or not the task frame is currently being accessed by another task. It is used to limit access to the stack frame control data to a single processor at a time. This word is

currently not accessed by any of the pseudo-machine instructions. This deficiency needs to be addressed.

14. Caller: When a called task executes an accept statement for a particular entry, a pointer to the base of the accepted caller is stored in this slot. This facilitates accessing the caller's top of stack to retrieve actual parameters during an entry call and also facilitates restarting the task upon completion of an accept body.

15. Return: This control word is used to record the return value of the program counter during a procedure call.

16. Entry: This word records the number of entries declared in the current activation, and is used to compute the amount of space required for entry frame control data. The instructions that use this control word assume that the entry frames are allocated immediately following the 'entry' control word. See paragraph 2.4.3.1 for an explanation of the structure and operation of the entry frame.

Program variables: The compiler allocates space for variables and objects after allocating space for the stack frame control data. Since the stack frame requires 16 words, the first word available for a local variable or object is the 17th word on the stack, if there are no task entries declared.

Temporaries: Temporaries are allocated as required during the evaluation of an expression. For example:

```
A := A + A
```

could translate to:

```
LOAD  A     -- The value of A is pushed on the stack
            -- thus allocating the first temporary.
LOAD  A     -- The value of A is pushed, now 2 temporaries
            -- are allocated.
ADD         -- The two operands are popped and the value
            -- of A + A is pushed.  Now, only 1 temporary
            -- is in use.
STORE A     -- The stack is popped, and the value is stored
            -- at the address for A. Now, no temporaries
            -- are in use.
```

LOAD, ADD and STORE are descriptive mneumonics and have the meaning described in their associated comments.

2.3.2 Stack processors

Each processor has 5 working registers: a program counter (PC), an instruction register (IR), a base register (BASE), a top of stack register (T), a heap pointer (HEAP) and a status register (STATUS).

1. PC The program counter is a pointer to words in the program memory. It indicates the next instruction that the processor will execute.

2. BASE The base register is a pointer to words in the stack memory. It indicates the first memory word of the stack frame of the currently executing subprogram or task.

3. IR The instruction register contains the instruction that the processor is currently executing.

4. HEAP The heap register is a pointer to words in the stack memory. It indicates the top of the dynamically allocated memory space.

5. T The top of stack register is a pointer to words in the stack memory. It indicates the top of the stack space of the currently executing subprogram

or task.

6. STATUS: The status register contains a bit that indicates whether or not the processor is busy or idle, and a field that points into the stack memory, called CURRENTJOB. This field must be the same size as the processor's other working registers, since it indicates the base of the currently executing task. The interpreter program requires an additional piece of information for multiple processor simulation, and this is also contained in the status register. This information is used to limit the number of instructions executed by a processor before it returns control to the supervisor. The supervisor then selects the next processor whose status is 'BUSY' and allows it to work on its assigned task. Thus, the supervisor serves to timeslice the actual processor among the simulated processors. For more information, see the code listing for the interpreter in appendix IV.

The following figure illustrates the configuration of a stack processor working on a task.

Note that the BASE, T and HEAP registers point into the stack space and indicate the first word in the stack frame, the top of the evaluation stack and the top of the dynamically allocated variable space, respectively. The instruction register holds the instruction that is currently executing, and the program counter points to the next instruction that will be executed. Finally, the STATUS register contains information necessary for multi-processing, and also points to the base of the task. Note that this diagram illustrates a task that has not called a subprogram, since the BASE register is equal to the

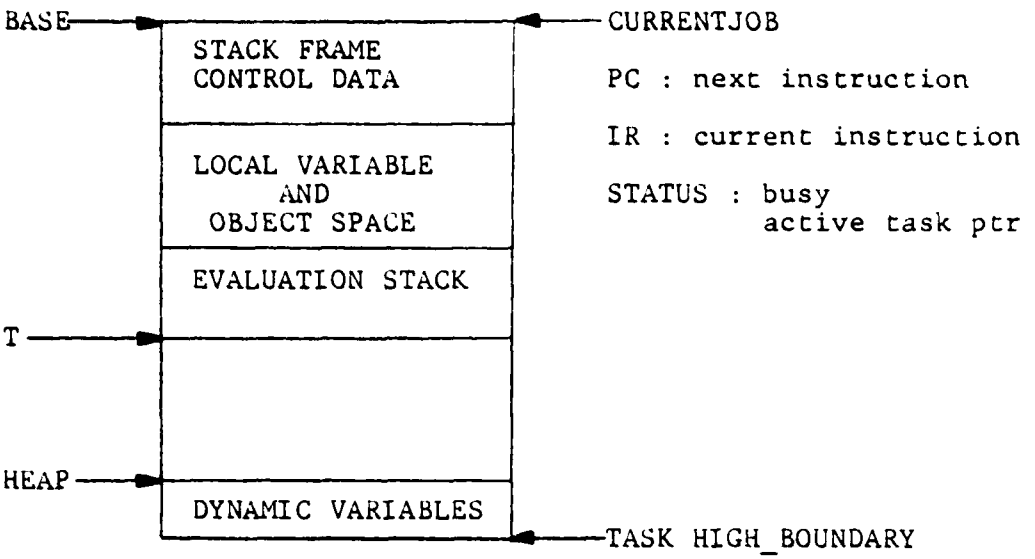


Figure 2-8: Sample Task Frame

base of the task object, marked by the CURRENTJOB register. If the task had called a single subprogram, the base register would point to the first word in the unused stack space (the first word of the new procedure activation), while CURRENTJOB would retain its position marking the base of the task.

Now that the architecture of the pseudo-machine has been discussed, the instruction set of the machine will be presented.

2.4 Pseudo-machine instruction set

The instruction set currently contains operations that are specifically tailored to a subset of the Ada language. The subset is described in the user's guide in appendix III. The currently implemented instructions can be divided into 5 classifications: relational operators, integer or single word arithmetic operators, tasking operators, I/O operators and miscellaneous operators. First, the instruction format will be described.

Each instruction contains three fields of information: the operation code field, the level field and the address field. The operation code field contains the name of the specific operation to be performed, and the level and address fields provide data necessary to perform that operation. If the level and address fields are not sufficient to contain the data required by the named operation, additional data words may follow that instruction. The following discussion of the operators does not specify the fields where such data is stored, but only lists the additional data required. Additional details may be obtained by reading the interpreter source listing in appendix IV.

2.4.1 Relational operators

The relational operators, EQUAL, GTR, GTREQ, LESS, LESSEQ, NOTEQ, ZXOR, ZAND and ZOR are binary operators which require no additional data for their operation. The two operands are assumed to reside on top of the temporary stack. (Boolean operands in the current implementation are not packed and require an entire word on the stack.) At run time, operands are popped from the stack and evaluated according to the indicated operation. A Boolean result is pushed back on the stack as defined by the following table.

S[T-1] A	S[T] B	EQUAL	GTR	GTREQ	LESS	LESSEQ	NOTEQ	ZXOR	ZAND	ZOR
0	0	1	0	1	0	1	0	0	0	0
0	1	0	0	0	1	1	1	1	0	1
1	0	0	1	1	0	0	1	1	0	1
1	1	1	0	1	0	1	0	0	1	1

Figure 2-9: Truth Table for the Binary, Relational Operators

ZNOT is a unary operator in this classification. The operand must be Boolean, and it is assumed to reside on top of the stack. ZNOT pops the stack and pushes the operand's opposite Boolean value.

S[T] A	ZNOT
0	1
1	0

Figure 2-10: Truth Table for the Unary-Operator ZNOT

2.4.2 Integer (single word) arithmetic operations

This classification contains single-word loads and stores, the binary operations, +, -, /, * and MOD, and the unary operator NEGATE.

2.4.2.1 Single word loads and stores

Operators exist for loading and storing variables and for loading constants.

Single-word loads and stores: The additional data required for load and store operations is the location of the variable whose value is to be loaded or the destination of the variable whose value is to be stored. As previously stated, the compiler cannot generate a 'hard' run-time address for a variable at compile time. However, since the storage space required by a single activation of a procedure is known, variables can be assigned locations relative to the beginning of the procedure's stack frame. Thus, at run time, a variable can be specified by providing its offset from the base. However, there is one complication.

Ada's visibility rules, as those of most other block structured languages, allow variables stored in other procedure activations to be accessed by the active subprogram. Thus, it is also necessary to specify which "base" the offset is relative to. The compiler provides this information as additional data with the instruction.

The single word loads and stores include the operators ILOAD and ISTORE. The operator ILOAD uses the additional data provided with the instruction to retrieve a variable from the variable storage area. Then it pushes the value on the evaluation stack. The operation ISTORE pops the stack and stores the value in the location specified by the additional data.

Loading constants: ILOADCONST is the operator for loading a constant value onto the evaluation stack. The additional data required by ILOADCONST is the value of the constant to be loaded. ILOADCONST takes the specified value and pushes it on the stack.

2.4.2.2 Arithmetic operators

The binary arithmetic operators + (IADD), - (ISUB), / (IDIV), * (IMULT), IMOD, and IREM assume that their operands reside on top of the stack. These operators all work similarly by removing two operands, applying the operation and pushing the result. The unary arithmetic operator, INEGATE, pops the stack and pushes the integer with the opposite sign.

2.4.3 Tasking operators

The tasking operators include ACTIVATE, CALLENTRY, ACCEPT, RELEASE, TERMINATE, ENTILOAD and ENTISTORE. Before considering the operation of these, a brief review of Ada's tasking facility might be desired. See section 2.2.2.1.

The additional data provided with these instructions is computed by the compiler at compile time. When compiling a task, the compiler computes the space required and allocates it within the local variable space of the task's parent. Other data, concerning the number of entries, priority and initial values for a task's HEAP, BASE, T, etc., are also computed and are available as additional information for the tasking operators.

2.4.3.1 ACTIVATE

The parent task executes the ACTIVATE instruction to initialize the stackframe of one of its nested tasks. Additional information provided with the instruction includes the nested task's base pointer, the heap pointer, a pointer to the task's code, the initial stack top, the task's priority and the number of entries in the task. The base, heap and initial stack top pointers are not absolute addresses but are relative to the base of the parent. The parent uses this information to compute absolute initial values for each of the processor's registers. The initial

values for the task's static and dynamic links are also set, and each entry declaration in the nested task is allocated space necessary for its control. This control data for the entry declaration is called an entry frame in the rest of this paper. The following paragraph describes the data it contains.

Three items of information are necessary for the control of an entry. The first item is a Boolean variable, referred to as the gate, which is used to record the status of the entry. If the task owning the entry is waiting to execute an accept statement for that entry, the gate is opened; otherwise, it is closed. The second item is a pointer to the code which is used to record the location of the code of the currently executing accept body. This is used only during the execution of a select statement, which has not been implemented in the translator. Currently, this item of information is not referred to in any of the implemented instructions. The final control word serves as a queue head pointer for tasks calling the entry. The pointer actually points to the stack frame of the first task waiting in the queue. Other tasks in the queue are chained together via the link field in their stack-frame control data.

In addition to allocating and initializing the entry

space, the parent initializes the rest of the stack frame and enters the nested task in the ready queue. After all tasks nested within the parent are initialized, the scheduler is called, and the tasks are assigned processors, if available.

2.4.3.2 CALLENTRY

The calling task executes the CALLENTRY instruction when it wishes to communicate with another task. Data included with the instruction includes the number of entries in the called task and the particular entry being called. First, the caller finds the appropriate entry frame in the called task's stack frame and enters itself in the wait queue for that entry. The caller then checks the entry gate to see if it is open or closed.

If the gate is open, the called task has previously executed an accept statement for this entry and found no callers waiting. (In response to this situation, the called task would have opened the gate, stored its context and released its processor. See paragraph 2.4.3.3, the ACCEPT operator.) In the pseudo-machine, the task owning the entry executes the accept body, so the calling task must awaken the suspended, called task. It does this by entering the task in the ready queue and by calling the scheduler. The caller also releases its processor, waiting for completion

of the accept body.

If the gate is closed, the called task is not waiting for a call to that entry. In this situation, the caller merely adds itself to that entry queue, and releases its processor.

2.4.3.3 ACCEPT

The called task executes the accept instruction when it is ready to communicate with a caller. Additional information necessary to execute an ACCEPT statement is the name of the entry being accepted. First, the appropriate entry frame is checked to see if any tasks are in the queue. If tasks are waiting, the first one is removed, and the appropriate accept body is executed. If no tasks are waiting, the called task opens the appropriate entry's gate, stores its processor's context, releases its processor and calls the scheduler.

2.4.3.4 RELEASE

The called task executes a release instruction after completing an accept body. RELEASE restarts the parallel execution of the calling task by returning it to the ready queue and calling the scheduler. No additional information is required by RELEASE, because a pointer to the caller is stored at a known offset in the stack-frame control data.

2.4.3.5 TERMINATE

A nested task executes a terminate instruction at the end of its execution. First, the terminating task notifies its parent that its execution is complete and checks to see if the parent was waiting for its termination. If so, the nested task enters its parent in the ready queue.

In either case, whether the parent was waiting or not, the nested task releases its processor and calls the scheduler. No additional information is required by TERMINATE since the nested task can locate the base of its parent, and since the parent's control data is stored at known offsets from its base. That is, TERMINATE is able to locate all the information it needs within the parent's stack frame.

2.4.3.6 ENTILOAD

The called task executes an ENTILOAD instruction only within an accept body when referencing a entry's formal parameter. The actual parameter corresponding to this formal parameter is retrieved from the caller's stack and is pushed on the called task's stack. Additional information required is the address of the actual parameter with respect to the calling task's 'T' register.

2.4.3.7 ENTISTORE

The called task executes an ENTISTORE instruction only within an accept body when assigning a value to an entry's formal parameter. A value is popped from the called task's stack and stored within the calling task's stack frame at the location of the corresponding actual parameter. Additional information required is the address of the actual parameter. This address is an offset with respect to the calling task's 'T' register.

2.4.4 I/O Operations

The input-output instructions include operators for writing strings (SPUT) and integers (IPUT) to the output file, and for reading integers (IGET) from the input file.

2.4.4.1 SPUT

Additional information required by the operator SPUT includes a line-feed Boolean that indicates whether or not a carriage return and a line feed is to be written on the output file, the number of characters to print and the character data itself. SPUT writes the indicated number of characters to the output file, and then, if the line-feed Boolean is true, it generates a carriage return/line feed.

2.4.4.2 IPUT

Additional information required by the operator IPUT is the line-feed Boolean. IPUT pops a word off the stack and writes the ASCII equivalent of the value to the output file. If the line-feed boolean is true, it generates a carriage return/line feed.

2.4.4.3 IGET

IGET reads a string of ASCII digits, delimited by a blank, from the input file, converts them to an integer value and pushes it on the evaluation stack. No additional information is required.

2.4.5 Miscellaneous Instructions

The miscellaneous instructions include operators to call a subprogram or function (CALL), to shift actual parameters in preparation for a function call (PARAMSHIFT) and to return from a call (RETURN). Other operators in this category include the absolute and conditional jumps (JMP, JMPF, JMPT) and an operator to increment the T register (INCT).

2.4.5.1 CALL

The calling subprogram executes a CALL instruction to set up an activation record for the called subprogram. The instruction initializes the static and dynamic links, stores the proper return address and initializes other information

within the stack frame. Additional information required is data to set the static link and a pointer to the code for the called subroutine.

2.4.5.2 PARAMSHIFT

Additional information required by the PARAMSHIFT operator is the number of parameter words to shift and the shift distance. The calling subprogram executes this instruction only when calling a procedure to allocate space on its evaluation stack for the return variable. The actual parameters are shifted upward on the stack the number of spaces indicated by the additional information.

2.4.5.3 RETURN

The called subprogram executes a RETURN instruction after completing its execution. If the called subprogram has no active nested tasks, it deallocates its stack space by resetting the T and Base registers and loads the return address into the program counter. If nested tasks are still active, the called subprogram cannot return; so it stores its context, releases its processor and calls the scheduler. No additional information is required to execute a RETURN instruction.

2.4.5.4 JMP

Additional data provided with the JMP operator is the destination address. JMP merely loads this address into the program counter so that the next instruction executed will be the one specified in the instruction.

2.4.5.5 JMPF, JMPT

Additional information provided with the conditional jump operators is the destination address. Both JMPF and JMPT pop a single operand from the stack and test its Boolean value. If the operand is false, JMPF loads the program counter with the destination address so that the next instruction executed will be the one specified. JMPT does just the opposite, transferring control only if the Boolean value is true.

2.4.5.6 INCT

Additional information provided with the instruction is the number of words to increment the T register. INCT adds the number provided to the current value of the T register.

This completes the description of the pseudo-machine's instruction set and also completes the description of the over-all design of the pseudo-machine. If more detailed information is desired, please refer to the interpreter listing in appendix IV. Now the project's second major product, the Ada test compiler, will be discussed.

2.5 The compiler

A compiler must recognize high level language constructs and translate them into equivalent machine level instructions. This section first considers the problem of translation and then describes the recognizer used in the Ada test compiler. Finally, the semantic routines that accomplish the translation are discussed.

2.5.1 Background -- Compilation

Before describing the project's test translator, a brief introduction to the compilation process will be presented. The emphasis will be on the problem raced by the translator, rather than on how the translation is specifically accomplished. This problem will be described by postulating the existance of a simple machine and a high level language and by using these tools to illustrate the compilation task.

The postulated machine: The postulated machine is a stack oriented machine that performs operations on operands previously placed on a stack. The machine described here is actually a subset of the PL/0 machine described by Niklaus Wirth in his book Algorithm's + Data Structures = Programs (Ref 14 : 331-336). Briefly, the machine's instructions are stored in program memory and are executed in sequential order unless the order is modified by an instruction. The

instruction set consists of the following 7 instructions.

1. LOD A: LOD A places the variable named 'A' on the stack.
2. LDC X: LDC X places the value 'X' stored in the instruction on the stack.
3. STO A: STO A saves the variable named 'A' in memory.
4. ADD: ADD removes two operands from the top of the stack and adds them together. The result is pushed on the stack.
5. CHECK <: CHECK < removes two operands from the top of the stack. If the second operand removed is less than the first, then the value TRUE is pushed, otherwise, the value FALSE is pushed.
6. JMP X: JMP X causes the machine to execute the instruction at location X next.
7. JMPF X: JMPF X removes an operand from the top of the stack. If its value is FALSE then the machine executes the instruction at location X.

The high-level language: The postulated high-level language consists of the single sentence:

IF <CONDITION> THEN <STATEMENT> ELSE <STATEMENT>.

The three words IF, THEN and ELSE give this sentence its structure. The word IF signals that a conditional statement will (or should) follow, and the words THEN and ELSE signal that a statement follows. Furthermore, the statement following THEN is to be done only if the conditional statement is true, and the statement following

ELSE is to be done only if it is false.

The translation: The specific example to be translated to hypothetical machine code is:

IF A < B THEN A := A + 1 ELSE B := B + 1.

In this sentence, A and B are variables that are assumed to have been initialized to some value, ':=' is an assignment operator, '+' is an addition operator and '1' represents the decimal number one.

The compiler's translation problem is similar to the problem faced by a human interpreter. The interpreter must take a sentence in the source language and create a sentence with the same meaning in the target language. Similarly, the compiler program must translate the meaning of a sentence written in a high-level language to a sentence with the same meaning in the machine's language. However, there is a difference between this translation problem and the human interpreter's problem. When translating between human languages, the interpreter is usually working with two languages of approximately the same expressive power, where a sentence in one language will become an equivalent sentence in the other language. In contrast, the compiler is working with two languages with vastly different expressive powers, where a sentence in a high-level language

may translate to hundreds of sentences in machine language. Thus, the compiler's goal is to provide a translation which preserves the meaning of a high level language sentence given the limited set of resources at the machine level.

There are several ways that a compiler can recognize a high level language construct, but these methods will not be discussed here. It is merely assumed that the compiler can recognize one. Once the construct is recognized, the associated meaning is known, and the compiler can issue machine language instructions which preserve that meaning. The sequence of code the compiler would generate to preserve the meaning of the statement

```
IF A < B THEN A := A + 1 ELSE B := B + 1
```

is:

```
   LOD A            --load variable A
   LOD B            --load variable B
   CHECK <          --remove A and B from the stack
                    --and replace with the value of A < B
   JMPF, FALSE_PT   --if the value on top of the stack is
                    --false, then go to label FALSE_PT
   LOD A            --begin true part
   LDC 1            --load the constant value 1
   ADD              --pop the two operands and push the sum
   STO A            --store the top of stack value in
                    --the location assigned to variable A
   JMP END          --jump over the false part
<<FALSE_PT>>
   LOD B            --begin the false part
   LDC 1            --load the constant value 1
   ADD              --pop two operands and push their sum
   STO B            --store the top of stack value in
                    --the location assigned to variable B
<<END>>             --end of translation
```

The reader should verify that this translation is correct. That is, assure that the defined meaning of the high level language construct is preserved in the translation to machine instructions. Now, with this example as background, the Ada to pseudo-code translator developed in the project will be described. The first topic to be discussed is the mechanism that recognizes high level language constructs, the LR(1) parsing automaton.

2.5.2 LR(1) parsing automaton

The LR(1) parsing automaton is a bottom-up, finite-state machine whose operations are directed by a set of language specific tables. For an introduction to LR(1) parsing see Appendix II.

The specific system used to build the parser was the LR package from Lawrence Livermore Laboratory (Refs 12 ; 13). This system is written in ANSII standard FORTRAN and consists of an automatic parser generator and a parser skeleton. Since the project was written in PASCAL, the parser skeleton had to be translated, and the tabular data output from the automatic parser generator had to be reformatted. The following paragraphs describe the construction of the parser, its structure and its operation.

2.5.2.1 Construction

Construction of the parser required inputting an LR(1) grammar into the automatic parser generator, inserting the resulting tables into the parser skeleton, and writing a lexical analyzer for Ada.

The automatic parser generator: The automatic parser generator constructs the language specific tables that control the operation of the automaton. An LR(1) grammar for the subject language, in this case Ada, is input to the generator, and a grammar analysis and a set of tables are produced. The grammar analysis consists of a sorted listing of the vocabulary, a formatted listing of the language productions and a human readable version of the resulting finite-state control for the parser. The set of tables is a machine readable version of the finite-state control and is in the proper format for insertion into the parser skeleton.

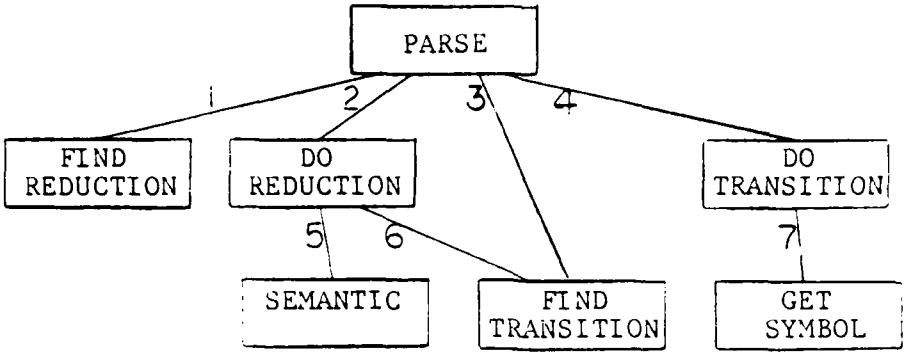
The parser skeleton: The parser skeleton is also written in FORTRAN 66 and consists of a set of routines that interpret the tables generated by the automatic parser generator. The package consisting of the parser skeleton and the tables requires the addition of a lexical analyzer to produce an operating parser. The lexical analyzer (scanner) scans the input file, isolates tokens and returns that token's reference to the parser. Collectively, the

parser skeleton, the tables and the scanner comprise the core of the compiler program.

Advantages: Using the LR system provides three major advantages. First, the parser can be constructed quickly with the assurance that the final results will parse the grammar correctly. Second, the grammar can be changed relatively easily, if necessary. Finally, the resulting program is space efficient and modular. The following paragraph describes the program's modular structure.

2.5.2.2 Parser structure

The following figure contains a chart which describes the structure of the parser using a technique described by G. L. Myers in his book Composite Structured Design (Ref 11 : 13). The figure consists of a network of modules arranged in levels, with a module's position determined by the calling dependencies between it and the other modules. The module at the top of the diagram is named PARSE. Module PARSE calls four subordinate modules, named FINDREDUCTION, DOREDUCTION, FINDTRANSITION, and DOTRANSITION. When PARSE calls FINDREDUCTION, it provides module FINDREDUCTION with information concerning the current state and the current look-ahead symbol. Upon return, FINDREDUCTION provides PARSE with a production number. These data flows are indicated on the diagram by a number, and the specific data



	IN	OUT
1	Current state and token	Production #
2	Production #, Stkptr	New state #, Stkptr
3	Current state and token	New state #
4	New state #, Stkptr	Stkptr, Current state # Token description
5	Production #, Stkptr	----------
6	Stack[stkptr].state, Left-hand side (production #)	New state #
7	----------	Token description

Figure 2-11: Parser-Structure Chart

passed is found in the chart at the bottom of the figure. The remainder of the diagram may be interpreted in this same way.

2.5.2.3 Parser operation

The controlling module, PARSE, repeatedly executes a series of statements until it transitions to its final state. The following algorithm describes PARSE's actions.

To begin, PARSE calls FINDREDUCTION to see if any reductions exist. If a reduction can be done, module DOREDUCTION is called, and PARSE goes back to repeat the

```
REPEAT
Check to see if a reduction is possible while in the
 current state with the current look-ahead token.
If reduction is possible: Do the reduction.
If reduction is not possible:
    Check to see if a transition is possible while in
      the current state with the current look-ahead token.
    If transition is possible:  Do the transition.
    If transition is not possible: Syntax error in source.
UNTIL current state = final state.
```

Figure 2-12: Algorithm for Module Parse

loop. If no reduction can be done, PARSE calls FINDTRANSITION to see if any transitions exist. If a transition can be done module DOTRANSITION is called and PARSE goes back to repeat the loop. If no transition can be done, PARSE has detected a syntax error. This sequence continues until the parser transitions to the final state.

2.5.3 Semantic routines

This section traces the flow of semantic information throughout the translation process. Semantic information is initially collected by the scanner and stored on semantic stacks. This information may eventually be transferred to the symbol table as directed by the semantic routines.

2.5.3.1 Scanner

SCANNER's function is to find the next lexical item in the input file. After finding it, the scanner also associates a certain semantic meaning with that token. For example, an identifier is returned to the parser as

(*identifier*, pointer to symbol table entry, ASCII representation). For parsing purposes, the only significant information is that the next token is an *identifier*. The semantic routines use the remainder of the information to determine whether or not this identifier is correctly used in the particular context.

2.5.3.2 Semantic stacks

The parser maintains 2 parallel stacks, one to store the next token and another to store the current state. In addition to these, various other stacks are maintained to store data associated with the stacked token. These stacks are called the semantic stacks. For example, additional semantic stacks can be used to store a pointer to the token's symbol table entry, to store its ASCII representation, its integer, real or character value or to retain any other information that might be required to establish the token's meaning. The semantic routines then use this information to determine if the token is proper for the given context.

2.5.3.3 Sample semantic routine

When DOREDUCTION calls SEMANTIC, it tells SEMANTIC which particular construct it has recognized in the input file. For example, assume that module DOREDUCTION has called SEMANTIC with production number 289. This tells the

semantic module that production number 289 has been isolated in the input and that the production's components have been assembled on the stack. Suppose production number 289 is defined as follows:

```
<PARAMETER_DECLARATION>  ::=  <ID>  :  <MODE_OPTION>
    <SUBTYPE_INDICATION>  <INITIALIZATION_OPTION>.
```

From this, the semantic module knows that the items composing a <PARAMETER_DECLARATION> are on top of the stack. Furthermore, it knows these items have been assembled on the stack by the parser in the order they were encountered; thus, the top item is an <INITIALIZATION_OPTION>, and the other items can be located by their offset from that item. Now, the appropriate semantic actions for a <PARAMETER_DECLARATION> can be accomplished using the data assembled in the semantic stacks. Appropriate semantic actions for a <PARAMETER_DECLARATION> might include:

- Look up the identifier (stack [stkptr - 4]) in the symbol table and check whether or not it has been previously declared.
- If it has been previously declared, call the error routine.
- If it has not been previously declared, add the new identifier to the symbol table, and store semantic data associated with it. e.g. Set the identifier's type to 'parameter', and store the parameter's mode (stack[stkptr - 2]), subtype (stack [stkptr - 1]) and initial value (stored at stack [stkptr]) in the symbol table.

After completing the semantic actions associated with production 289, SEMANTIC returns control to DOREDUCTION. DOREDUCTION then removes the five items comprising production 289 from the stack and replaces them with the single item <PARAMETER_DECLARATION>.

2.5.4 Symbol table and visibility

Name visibility is enforced with a compile time environment stack, stacking rules and special symbol-table access routines.

2.5.4.1 Environment stack

An entry in the environment stack contains information on the name of the environment, whether or not the environment acts as a package visible part, and whether or not the environment is directly visible. Initially, the environment stack is empty.

2.5.4.2 Stacking rules

The stacking rules specify which names are to be pushed on the environment stack and what values are to be stored with them. These rules assume the existence of two operations on the environment stack, PUSH and POP, and of a global variable used to record the lexical level. Again, the environment stack consists of 3-tuples which contain the environment name and two Boolean variables that indicate whether or not the environment acts like a package visible

part and whether or not the environment represented by the entry is directly visible.

```
PROCEDURE
  entry :  increment the lexical level;
           PUSH (procedure name, false, true);

  exit  :  decrement the lexical level;
           POP until name = procedure name
           POP

PACKAGE VISIBLE PART
  entry :  PUSH (package name, true, true);

  exit  :  POP until name = package name
           REPLACE (package name, true, false)

PACKAGE BODY
  entry :  PUSH (package name, true, true)

  exit  :  POP until name = package name
           POP

TASK VISIBLE PART
  entry :  PUSH (task name, true, true)

  exit  :  REPLACE (task name, true, false)

TASK BODY
  entry :  increment the lexical level
           PUSH (task name, false, true)

  exit  :  decrement the lexical level
           POP until name = task name
           POP

ACCEPT BODY
  entry :  PUSH (entry name, true, true)

  exit  :  POP
```

Figure 2-13: Stacking Rules

2.5.4.3 Symbol-table routines

Ada's visibility rules are supported by routines to enter a symbol, to find a symbol in a named environment and to find a symbol in scope.

Entering a symbol

Symbols are entered in the symbol table tagged with the environment in which they are declared. This environment is specified by a 2-tuple consisting of the lexical level and a linked list of the environment names on the stack when the entry is made.

Lexical Level: The lexical level records the number of static links that must be traversed to reach the main or outermost textual level. This value is initialized to 0 and altered only according to the stacking rules.

Linked list: The linked list contains all the directly visible names stored on the environment stack at the time the symbol was entered. Since the environment stack is altered only according to the stacking rules, the list will contain only subprogram, package, task or entry names.

Finding a symbol in a named environment

The caller provides the symbol's lexical level and the specific environment to be searched. The routine searches this environment and returns a reference to the symbol if it

exists.

Finding a symbol in scope

The caller provides the current lexical level and environment. The routine successively searches nested scopes until the symbol is found or there are no more environments to search. It returns a reference to the symbol if it exists.

2.5.4.4 Visibility example

These tools comprise a system which supports Ada's visibility rules. The following example illustrates their use.

```
(1)
(2) PROCEDURE MAIN IS                    *
      A : INTEGER;
(3)
     PACKAGE MAIN_1 IS                   *
(4)    A : INTEGER;
       END MAIN_1;                       *
(5)
       PACKAGE BODY MAIN_1 IS            *
(6)      M_1B : INTEGER;
       BEGIN
         A := 1;
         MAIN.A := 2;
       END MAIN_1;                       *
(7)
   BEGIN -- MAIN
(8)  MAIN_1.A := A;
   END MAIN;                             *
(9)
```

Figure 2-14: Example Program for Visibility Demonstration

Each number on the figure points to a region of the text where the environment is of interest, and the asterisk

marks the application of one of the stacking rules. For each number, figure 2-15 illustrates the contents of the environment stack and all the variables entered in the symbol table up to that point in the source text. The example begins at point one, with a NIL environment and no variables in the symbol table. Each new identifier encountered in the text is entered in the current environment, and the environment stack is changed only at the marked points using the previously-defined stacking rules.

This concludes the description of the thesis project. Additional detailed information on the operation of either the pseudo-machine or the compiler can be obtained by studying the PASCAL source listing for the system. Appendix IV contains the listing of the interpreter program, but due to the size of the compiler listing, it has not been included. However, copies of the entire listing are available in machine readable form on the ARPA net. Contact the AFIT/EN Mathematics Department for further information. The following chapter will describe recommendations for follow-on efforts.

	CONTENTS OF ENVIRONMENT STACK			VARIABLES ENTERED	
	ENV NAME	PACKAGE	DIRECT VIS	VAR NAME	ENVIRONMENT
1	NIL	NIL	NIL	NIL	NIL
2	MAIN	FALSE	TRUE	MAIN	0 <NIL>
3	MAIN	FALSE	TRUE	MAIN	0 <NIL>
				A	1 <MAIN>
4	MAIN	FALSE	TRUE	MAIN	0 <NIL>
	MAIN_1	TRUE	TRUE	A	1 <MAIN>
				MAIN_1	1 <MAIN,MAIN_1>
				A	1 <MAIN,MAIN_1>
5	MAIN	FALSE	TRUE	SAME AS ABOVE	
	MAIN_1	TRUE	FALSE		
6	MAIN	FALSE	TRUE	MAIN	0 <NIL>
	MAIN_1	TRUE	FALSE	A	1 <MAIN>
	MAIN_1	TRUE	TRUE	MAIN_1	1 <MAIN>
				A	1 <MAIN, MAIN_1>
				M_1B	1 <MAIN, MAIN_1>
7	MAIN	FALSE	TRUE	SAME AS ABOVE	
	MAIN_1	TRUE	FALSE		
8	MAIN	FALSE	TRUE	SAME AS ABOVE	
	MAIN_1	TRUE	FALSE		
9	NIL	NIL	NIL	NIL	NIL

Figure 2-15: Visibility Rules Demonstration

3. Recommendations

As time for the project work drew to a close, it became apparent that several items on the 'do-list' would not get done. This chapter describes these deficiencies and also describes some areas where continuation efforts could begin. Since the project is composed of two major parts, the pseudo-machine and the test compiler, the recommendations are divided to reflect this. The first section in the chapter describes suggested improvements to the pseudo-machine, and the last section describes suggested improvements to the test compiler.

3.1 Improvements to the pseudo-machine

Known areas where the pseudo-machine can be improved or expanded include: providing run-time space allocation for tasks, improving the system queues, improving the allocation of stack-frame control data, implementing exceptions, implementing the dynamic-variable-space-access routines, investigating the effects of Ada's enumeration I/O requirements and implementing mechanisms to protect data subject to access by multiple processors.

3.1.1 Run-time space allocation

The current implementation computes a task's space requirements at compile time. Thus, if a task calls procedures that recurse excessively, the precomputed space

may become exhausted. If this is unacceptable for a particular implementation, consideration should be given to a run-time space allocation scheme.

3.1.2 System queues

The ready and entry queues in the system are implemented as linked lists with a pointer to the head of each list. Therefore, adding a task to a queue requires traversing the entire list to find the list's end. Possibly, the queues could be speeded up by adding a queue tail pointer, but the average queue length could be so short that this would not be much of an improvement.

3.1.3 Stack-frame control data

The current implementation uses the same stack-frame control data for tasks as it does for subprogram activations. This results in several unused words in the control data allocated to a subprogram. Some space could be saved by defining a new stack frame specifically for use in a subprogram call.

3.1.4 Implementing exceptions

Exceptions declared in a block or subprogram must be allocated space for control information. Data, such as the names of the exceptions handled within the block and the location of the handler's code, must be available at fixed or computable offsets from the BASE register. An

instruction to carry out the run-time actions of raising an exception must also be written.

3.1.5 Implementing dynamic variables

No constructs that required dynamically allocated heap space were implemented. However, when they are, run-time actions that will be required will include instructions to load dynamic variables onto the evaluation stack, to store the top of stack within the dynamic variable space, and to locate and manipulate data within dynamically created task objects. Once methods for allocating, loading and storing dynamic variables and tasks have been implemented, consideration should be given to deallocation and garbage collection. However, a minimal system should not require this.

3.1.6 Enumeration I/O

Ada's enumeration I/O facilities may require the addition of another data block to the stack frame and the addition of another register with which to access it. This area may be necessary to hold the ASCII representation of enumeration types declared within the associated scope.

3.1.7 Data protection

The data-lock control word was added to the control data so that access to a task's stack frame could be limited to a single processor at a time. However, since the pseudo-machine architecture was simulated on a single processor, a mechanism for checking and setting the data lock was never implemented. Currently, none of the implemented instructions check this word, although several of them should. In addition to protecting data contained in a task, the system's ready queue must be similarly protected. This entire matter requires careful consideration.

3.2 Improvements to the compiler

Improvements to the test compiler must be based on its intended use. This section considers two possible uses of the test compiler: first, as a basis for building a finished compiler and, finally, as a tool in the development of a production Ada to pseudo-code compiler.

3.2.1 Towards a finished product

Several compilation tasks were side stepped in this project because of time restrictions. Four of these areas include representation specifications, types, overloading and separate compilation. Representation specifications specify how types in the language are to be mapped onto the

underlying machine (Ref 2 : 13-1). This issue was not investigated. Implementation of types was limited to integers, only. The addition of user defined types will greatly increase the power of the compiler and should not be overly difficult. However, implementing subtypes and derived types could be more sporting. Overloading and separate compilation are two interesting and probably very challenging areas that will likely have a significant impact on the structure of the test compiler's symbol table. Separate compilation will have an additional impact on its code generation routines and will probably require a comprehensive linker program.

As the compiler moves closer to completion, more thought should be given to improving the error tolerance of the semantic routines and to polishing the grammar. Currently, the compiler checks for syntactic and semantic errors until the first error is encountered. From that point onward, only syntactic errors are checked. More error tolerant semantic routines would allow continued analysis of semantics after an initial error. The LR(1) grammar used in the test compiler's parser was originally obtained from Intermetrics and has been slightly modified so that it is more suitable for a one pass compiler. However, since several productions are unused, the grammar can be streamlined further. This would result in minor increases

in execution speed and in minor decreases in memory space requirements.

The following deficiencies in the current compiler have been noted.

- Accept statements: Accept statements for an entry of a given task may only appear within the sequence of statements of the corresponding task body (Ref 2 : 9-7). The compiler does not check for this restriction.
- Package body variables: Variables declared in a package body should not be accessible outside the package. The compiler does not limit such access.
- Initialization of variables: The compiler does not handle initialization of variables.
- Testing: The compiler program has not received adequate testing because of time limitations, and some of the implemented constructs have not been tested at all. Be assured that there are errors to be found.

3.2.2 For use as a tool

With minor improvements, the existing test compiler and pseudo_machine could be used as a tool to begin the development of a production-quality Ada to pseudo-code translator. The major improvement necessary to make the test-compiler useful for this purpose is the implementation of the basic structured types. In this category, records and one dimensional arrays are almost necessities. Additional constructs that should be added are enumeration and access types. With these additions, sufficient power

should be available to write the new compiler in Ada without overly limiting one's expression.

When writing the production compiler, the programmer must deal with the limitations of the host processor. For example, if the compiler is to run on a small machine, it must be designed with this in mind. This means that space saving techniques such as segmentation and multiple-pass compiler design would probably have to be employed.

This completes the Recommendations chapter and also the main body of the thesis. Pursuing this project has added a staunch supporter to the growing ranks of Ada enthusiasts, and I feel that Ada is something that has been needed for a long time. More power to her!

BIBLIOGRAPHY

1. Barrett, William A. and John D. Couch. Compiler Construction Theory and Practice. USA: Science Research and Associates, Inc., 1979.

2. Defense Advanced Research Projects Agency. Reference Manual for the Ada Programming Language, Proposed Standard Document. Washington, D.C.: Department of Defense, 1980.

3. ----- Requirements for Ada Programming Support Environments, STONEMAN. Washington, D.C.: Department of Defense, 1980.

4. Fisher, D.A. A Common Programming Language for the Department of Defense -- Background and Technical Requirements. Arlington, Virginia: Institute for Defense Analyses, Science and Technology Division, 1976. (AD A028 297).

5. Fox, Joseph M. Benefit Model for High Order Language. McLean, Virginia: Decisions and Designs, Inc, 1978. (AD A053 032).

6. Harrison, Michael A. Introduction to Formal Language Theory. Reading, Massachusetts: Addison and Wesley Publishing Company, 1978.

7. Habermann A. N. and Isaac R. Nassi. "Efficient Implementation of Ada Tasks". Pittsburgh, Pa: CMU-CS-80-103, Carnegie-Mellon University, 1980.

8. Honeywell, Inc. Formal Definition of the Ada Programming Language, Preliminary Version for Public Review. Minneapolis: Systems and Research Center, 1980.

9. Ichbiah, J.D. and others. Rationale for the Design of the Ada Programming Language. New York: Association for Computing Machinery, Inc., 1979.

10. Institute for Information Systems. PASCAL System II.0 User's Manual. La Jolla, Ca: IIS, 1979.

11. Myers, Glenford J. Composite / Structured Design. New York: Van Nostrand Reinhold Company, 1978.

BIBLIOGRAPHY

12. Shannon, Alfred. The LR System. FORTRAN source listing for the LR system. Argonne, Illinois: National Energy Software Center, Version 61, 1979.

13. Wetherell, Charles and Alfred Shannon. "LR, Automatic Parser Generator and LR(1) Parser." Livermore, California: Lawrence Livermore Lagoratory, 1979.

14. Wirth, Niklaus. Algorithms + Data Structures = Programs. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1976.

15. Wirth, Niklaus. PASCAL S. Source listing for PASCAL S. Zeurich: Institut Fuer Informatik, 1976.

1 APPENDICES

These appendices provide additional detailed information on several topics. Included in them are a summary of an early study on the economics of commonality, background information on LR(1) parsing, a user's guide and the PASCAL source listing of the interpreter program.

I. DOD Commonality study

In July 1977, DARPA tasked Decisions and Designs Incorporated (DDI) to perform a two part effort: first, to modify decision analytic models to predict the impact of a common DOD high order language, and second, to implement and run the models (Ref 5 : 1).

Decision Analytic models:

Three models were used to accomplish this task: EVAL, which compared 14 attributes of the input languages, SPREAD, which generates predictive scenarios given data inputs from EVAL and other sources, and DECISION, which shows the effects of different decisions given the scenarios generated by SPREAD and event probabilities estimated by the user.

Implementation:

The table on the following page summarizes the results of the study.

Each column in the table represents a different scenario. For example, column I illustrates the effect of introducing DOD (the generic name for the proposed common language) in 1980 and achieving total acceptance of the language by 1985. (Total acceptance is defined as the point where all contracts for new software are to be written in DOD.) Programs written in other languages are assumed to continue throughout their life cycles without rewrite. Each

	I	II	III	IV	V	VI	VII	VIII	IX 4 Lang. DoD	X
	1980-85	1985-90	1980-90	1981-86	1983-88	1987-92	1982-83	1982-85	1980-85	No DoD
1977	487	487	487	487	487	487	487	487	487	490
1978	476	476	476	476	476	476	476	476	476	479
1979	483	482	482	482	482	482	482	482	482	486
1980	638	637	637	637	637	637	637	637	637	642
1981	676	654	654	654	654	654	654	654	654	657
1982	707	652	674	674	652	652	652	652	645	653
1983	761	493	612	616	493	493	820	588	557	511
1984	917	516	678	740	533	516	987	690	598	537
1985	1,064	553	776	861	601	553	1,113	811	641	576
1986	1,399	629	976	1,113	656	620	1,558	1,037	786	651
1987	1,561	679	1,097	1,294	787	637	1,783	1,218	839	675
1988	1,775	809	1,242	1,511	1,017	700	2,065	1,500	934	701
1989	1,691	987	1,301	1,570	1,288	749	1,749	1,610	872	620
1990	1,774	1,208	1,461	1,686	1,500	867	1,781	1,726	911	628
1991	2,016	1,554	1,739	1,980	1,827	1,087	2,024	1,969	1,021	656
1992	2,363	1,934	2,100	2,326	2,214	1,386	2,373	2,316	1,180	738
1993	2,511	2,162	2,295	2,472	2,409	1,633	2,520	2,461	1,261	762
1994	2,554	2,313	2,427	2,512	2,450	1,854	2,563	2,502	1,309	778
1995	2,385	2,237	2,293	2,341	2,275	1,968	2,396	2,330	1,244	758
1996	2,333	2,278	2,235	2,285	2,217	2,020	2,345	2,273	1,256	763
TOTAL	28,571	21,740	24,642	26,737	23,677	18,471	29,473	26,419	16,790	12,761

YEARLY SAVINGS: 1st Year is Year of Introduction
2nd Year is Year in which all new starts are in DoD; except for CASE IX, where the second year is year in which all new starts are divided among four languages, one of which is DoD.

row in the table represents a different year as labeled in the leftmost column.

The data in the table represents millions of dollars saved as compared to a baseline of exclusive use of assembly language. Thus, any model that considers the use of nearly any HOL will exhibit savings. For comparison purposes, column X models the current situation; that of "no-change" in DOD 5000.31 estimate (Ref 5 : 5). are estimated using a software expenditure of 3.2 billion dollars per year.

The data is provided so that the reader can make his own conclusions. The author stated that for a 5 year introduction period, delay of the introduction from 1980 to 1987 reduces savings by about 1.5 billion per year. He concluded with

> It is recommended that the DOD single common high order language be introduced as rapidly as possible without penalizing technical quality or acceptability... (Ref 5 : 8)

II. LR(1) Parsing automaton

An LR(1) parsing automaton is a machine that can recognize any sentence in a particular deterministic language, and conversely, reject any sentence not contained in that language. To define the machine, the term language will be defined, and a sample language introduced to illustrate the operation of the machine. Then, the components of the machine and their operation will be described.

A language consists of a collection of symbols, called an alphabet, arranged according to a set of rules. These rules are called productions, and the collection of all these rules, or productions, is called a grammar.

The alphabet of the sample language includes only the following three symbols : BREAD, EATS, and JOHN. The production rules which govern their placement are listed below.

```
1.  SENTENCE  ::=  SUBJECT  VERB  OBJECT
2.  SUBJECT   ::=  JOHN
3.  VERB      ::=  EATS
4.  OBJECT    ::=  BREAD
```

(The symbol '::=' means 'is defined as')

This grammar consists of four productions. Each production consists of two parts, a left-hand side and a right-hand side, separated by the symbol '::='. The number

of symbols on the right-hand side of a production is called the length of that production. For example, the length of production 1 is 3. Note that these productions introduce some new symbols. The symbols SENTENCE, SUBJECT, VERB and OBJECT do not appear in the alphabet of the language but are necessary to describe intermediate representations of the sequence being generated. Also note that only one of these new symbols does not appear on the right-hand side of a production. This symbol, SENTENCE, is called the start symbol of the grammar.

The start symbol is a representation of all the possible strings that can be generated by the grammar. In this case, the start symbol SENTENCE is defined as a SUBJECT followed by a VERB and then an OBJECT. Similarly, a subject is defined as JOHN, a VERB as EATS and an OBJECT as BREAD. Therefore, in this grammar, the start symbol represents the single string 'JOHN EATS BREAD'.

To recognize a string in a language, the automaton must reconstruct the particular sequence of derivations that began with the start symbol and resulted in the string. If the automaton accomplishes this successfully, the string is accepted as part of the language; otherwise it is rejected.

The machine contains four components, an input device, an output device, a memory device and a control module.

Input device: The input device consists of a tape containing the sentence to be checked, and a head to read the tape. The read head scans the sentence from left to right and provides the machine with one symbol at a time.

Output device: The output device consists of a blank tape and a write head. The machine uses the output tape to store a history of the productions used to analyze the sentence.

Memory device: The machine's memory device is a stack. Each time the machine accesses the stack, it stores two pieces of information. First, it stores a symbol from the grammar, and then it stores a table or its representation used to define the machine's next action. These two items are referred to as a data pair in the rest of this appendix.

Control: The final component to discuss is the control module. The control module directs the operation of the machine's only two functions which are shifts and reductions. It determines which of these instructions to execute by entering the table stored on top of the stack with the next symbol on the input tape as an argument. If the table indicates that a shift should be done, the control unit stacks the look-ahead symbol and the table whose name is stored with the shift instruction. It then advances the read head to the next symbol on the input tape. If the

table indicates that a reduction should be done, the control unit writes the indicated production number on the output tape and looks up the length of the production. It removes this number of data pairs from the stack and then consults the uncovered table. This time, instead of using the next symbol from the input tape, the control unit uses the left-hand side of the production to enter the table.

To accomplish shifts and reductions, the control unit must know the productions of the grammar and also the contents of the tables. Therefore, the structure of the control unit must include a representation of this data in some form. Assume that the control unit knows the productions which make up the sample grammar and also the contents of the tables illustrated in the following figure.

	SENTENCE	SUBJECT	VERB	OBJECT	JOHN	EATS	BREAD	⌐
T0	ACCEPT	SHIFT, T3			SHIFT, T2			
T1						REDUCE, 2		
T2								
T3			SHIFT, T6			SHIFT, T5		
T4								
T5							REDUCE, 3	
T6				SHIFT, T8			SHIFT, T7	
T7								REDUCE, 4
T8								REDUCE, 1

'⌐' is a special symbol indicating 'end of input'.

Figure 3-1: Tables for the LR(1) Parsing Automaton

The machine begins with table T0 on the stack and with

the input unit looking at the first symbol on the input tape. The machine will attempt to recognize the sequence of symbols 'JOHN EATS BREAD' as a legitimate string in the language specified by the sample grammar's production rules. The machine enters table T0 with the first symbol, JOHN, and finds the entry 'shift,T1'. On this shift move, the control unit stacks the input symbol, JOHN, and the new table found, T1, and then reads the next symbol, EATS. The current stack configuration is illustrated in the following figure.

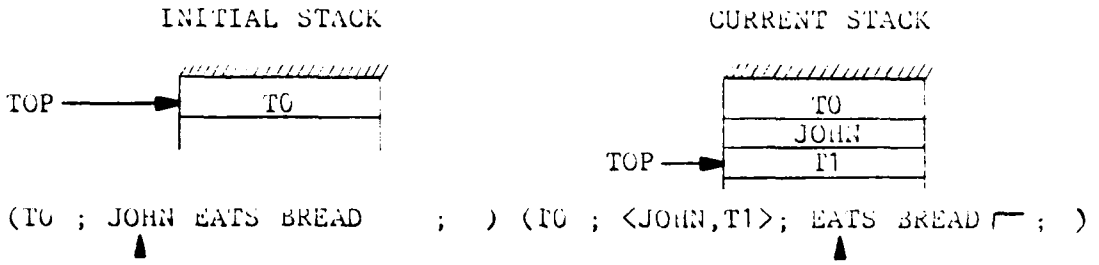
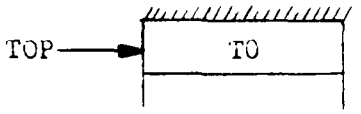


Figure 3-2: Result of the 'Shift, T1' Move

The symbols within the parenthesis on the diagram constitute an instantaneous description of the machine's state. The symbols before the first semicolon represent the items stored on the stack, the symbols between the semicolons represent the unprocessed portion of the input tape, and the symbols after the last semicolon represent the contents of the output tape. The left-most word of the input tape's representation, marked by ▲ in the diagram, is the parser's current look-ahead token. Collectively,

these items completely specify the machine's current configuration.

Presently, the items T0, JOHN and T1 are stacked and the look-ahead token is EATS. The control unit enters table T1 with the symbol EATS and finds the entry 'reduce,3'. On this reduction, the control unit writes the number 3 on the output tape and looks up the length of production 3. Production 3 is 1 symbol long, so the control unit removes 1 data pair from the stack, leaving the following configuration:

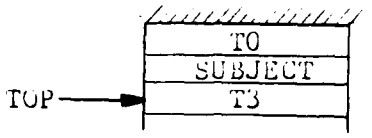


(T0 ; EATS BREAD ; 2)

Figure 3-3: Intermediate result of the 'Reduce, 3' move

The control unit enters the table on top of the stack with the lefthand side of production 3, which is the symbol SUBJECT. It finds the entry 'shift,T3'. The configuration of the machine after this move is illustrated in the following diagram.

The machine continues in this manner until it reaches 'ACCEPT' or it cannot do a transition or a reduction. If it reaches 'ACCEPT', the input string has been successfully



(T0 ; <SUBJECT,T3> ; EATS BREAD ; 2)

Figure 3-4: Final result of 'Reduce,3' move

parsed and, thus, is a part of the language specified by the grammar. If the machine cannot do a transition or a reduction, the input string is not part of the language and it is rejected. The following figure contains the instantaneous descriptions of the machine for every step required to recognize the string 'JOHN EATS BREAD'.

```
BEGIN
 (T0 ; JOHN EATS BREAD      ;  )
 (T0, <JOHN, T1> ; EATS BREAD     ;  )
 (T0, <SUBJECT, T3> ; EATS BREAD     ; 2)
 (T0, <SUBJECT, T3>, <EATS, T5> ; BREAD     ; 2)
 (T0, <SUBJECT, T3>, <VERB, T6> ; BREAD     ; 2, 3)
 (T0, <SUBJECT, T3>, <VERB, T6>, <BREAD, T7> ;     ; 2, 3)
 (T0, <SUBJECT, T3>, <VERB, T6>, <OBJECT, T8>;     ; 2, 3, 4)
ACCEPT
```

Figure 3-5: Acceptance of the String "JOHN EATS BREAD"

This concludes the description of the structure and operation of an LR (1) parsing automaton. Although its operation may seem overly complex, the automaton is well suited for computer implementation. In fact, such an automaton can be generated automatically by computer given

the grammar to be parsed (Ref 13). This greatly simplifies the construction of command languages and compilers.

III. User's guide

This appendix describes the input accepted by the test compiler and the output which results. Several example programs are also included.

Input: Input to the program should be an Ada text file whose constructs have been selected from the implemented subset. Language constructs that may be used to compose input programs are listed below.

1. Integer variables. Number declarations and variable initializations are not implemented.
2. Package declarations.
3. Procedures and functions with parameters (mode types may be specified)
4. Task declarations.
5. Selected components may be used to open visibility to objects that are within scope but which are not directly visible.
6. Most integer arithmetic or boolean expressions may be used including those using short circuit conditions. However, the following list of operators has not been implemented: REM, **, &, IN.
7. The following statements may be used:

    a. Assignment

    b. Procedure, function or entry calls

    c. Exit

    d. Return

    e. IF THEN ELSIF ELSE

AD-A100 796 AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOO--ETC F/6 9/2
PRELIMINARY DESIGN AND IMPLEMENTATION OF AN ADA PSEUDO-MACHINE.(U)
MAR 81 A R GARLINGTON

UNCLASSIFIED AFIT/GCS/MA/81M-1 NL

2 of 2
AD A 100796

END
DATE FILMED
7-81
DTIC

III. User's guide

This appendix describes the input accepted by the test compiler and the output which results. Several example programs are also included.

Input: Input to the program should be an Ada text file whose constructs have been selected from the implemented subset. Language constructs that may be used to compose input programs are listed below.

1. Integer variables. Number declarations and variable initializations are not implemented.
2. Package declarations.
3. Procedures and functions with parameters (mode types may be specified)
4. Task declarations.
5. Selected components may be used to open visibility to objects that are within scope but which are not directly visible.
6. Most integer arithmetic or Boolean expressions may be used including those using short circuit conditions. However, the following list of operators has not been implemented: REM, **, &, IN.
7. The following statements may be used:

    a. Assignment

    b. Procedure, function or entry calls

    c. Exit

    d. Return

    e. IF THEN ELSIF ELSE

f. Accept

g. loops (except FOR loop)

Output: The output of the program is dependent on a specially defined pragma. This pragma was added to allow more direct control of the program throughout its development. Its format is:

PRAGMA TOGGLE (<OPTION_STRING>),

where <OPTION_STRING> is composed of selections from the following list of options: EXECUTE, TRACESTORE, PRINTCODE, TRACEPARSE, TRACETOK. Multiple selections must be separated by commas.

All of these options are initially off. To select an option, list it in an option string, and the compiler's output will be as defined below:

EXECUTE: If no errors are detected in the input program, the program will be executed.

TRACESTORE: TRACESTORE will do nothing unless EXECUTE is also selected. If EXECUTE is selected, each value stored during the execution of an ISTORE or ENTISTORE command will be printed.

PRINTCODE: The code generated by the compiler is formatted and printed.

TRACEPARSE: Each transition or reduction made by the parsing automaton is printed. This listing is fairly long even for a short program.

TRACETOK: The representation of each token passed from the scanner to the parser is printed. This representation consists of the token's vocabulary index as output from the automatic parser generator (Ref 13).

The following examples illustrate the effects of selecting these options given a simple input program.

```
ADA-G COMPILER
AIR FORCE INSTITUTE OF TECHNOLOGY

   1 -- THIS EXAMPLE ILLUSTRATES THE COMPILER'S OUTPUT WITH
   2 -- NO CONTROL INFORMATION.
   3
   4 PROCEDURE MAIN IS
   5   A : INTEGER;
   6 BEGIN
   7   A := 3;
   8   PUT (" A = ");
   9   PUT_LINE (A);
  10 END MAIN;
```

```
ADA-G COMPILER
AIR FORCE INSTITUTE OF TECHNOLOGY

   1 -- NOW THE SAME PROGRAM IS INPUT TO THE COMPILER WITH
   2 -- THE EXECUTE OPTION SELECTED.
   3
   4 PRAGMA TOGGLE (EXECUTE);
   5
   6 PROCEDURE MAIN IS
   7   A : INTEGER;
   8 BEGIN
   9   A := 3;
  10   PUT (" A = ");
  11   PUT_LINE (A);
  12 END MAIN;
A =           3
```

```
ADA-G COMPILER
AIR FORCE INSTITUTE OF TECHNOLOGY

   1 -- OPTION PRINTCODE PRINTS THE CODE GENERATED BY THE COMPILER
   2 -- FOR THE INPUT PROGRAM.  THIS OPTION IS SELECTED IN THIS
   3 -- EXAMPLE.
   4
   5 PRAGMA TOGGLE (PRINTCODE);
   6
   7
   8 PROCEDURE MAIN IS
   9    A : INTEGER;
  10 BEGIN
  11    A := 3;
  12    PUT (" A = ");
  13    PUT_LINE (A);
  14 END MAIN;


*** PRAGMA PRINT_CODE ***

 INDEX   MNEMONIC   LEVEL   ADDRESS
    0   JMP            0         1
    1   INCT           0        18
    2   ILOADCONST     0         3
    3   ISTORE         0        17
    4   SPUT           0         5
    5   DATA           0        32
    6   DATA           0        65
    7   DATA           0        32
    8   DATA           0        61
    9   DATA           0        32
   10   ILOAD          0        17
   11   IPUT           1         0
   12   RETURN         0         0
```

```
 1 -- NOW THE TRACESTORE OPTION IS SELECTED.
 2 -- THIS OPTION WILL PRINT THE VALUE STORED DURING THE EXECUTION
 3 -- OF THE ISTORE INSTRUCTION (INSTRUCTION NUMBER 3 IN THE PREVIOUS
 4 -- EXAMPLE.
 5
 6 PRAGMA TOGGLE (EXECUTE, TRACESTORE);
 7
 8
 9 PROCEDURE MAIN IS
10   A : INTEGER;
11 BEGIN
12   A := 3;
13   PUT (" A = ");
14   PUT_LINE (A);
15 END MAIN;


*** PRAGMA TRACESTORE ***

EACH VALUE STORED DURING EXECUTION OF AN ISTORE COMMAND IS LISTED

          3
A =               3
```

```
ADA-G COMPILER
AIR FORCE INSTITUTE OF TECHNOLOGY

 1 -- NOW THE RATHER LENGTHY OUTPUT GENERATED BY THE TRACEPARSE
 2 -- OPTION IS DEMONSTRATED.  EACH TRANSITION OR REDUCTION
 3 -- MADE BY THE PARSING AUTOMATON IS PRINTED.  TO LIMIT THE
 4 -- LENGTH OF THE OUTPUT, A SHORTER PROGRAM IS INPUT AS FOLLOWS:
 5 --
 6 --                 PROCEDURE MAIN IS
 7 --                 BEGIN
 8 --                   NULL;
 9 --                 END MAIN;
10 --
11 -- AS YOU WILL SEE, THE PARSER IS VERY BUSY EVEN WITH A SIMPLE
12 -- EXAMPLE LIKE THIS.
13
```

```
  14
  15 PRAGMA TOGGLE (TRACEPARSE);
PRODUCTION 18 AND TRANSITION FROM STATE  2 TO STATE  7
TRANSITION FROM STATE  7 TO STATE 41
  16
  17 PROCEDURE MAIN IS
PRODUCTION 16 AND TRANSITION FROM STATE  2 TO STATE  8
PRODUCTION 15 AND TRANSITION FROM STATE  2 TO STATE  9
PRODUCTION 385 AND TRANSITION FROM STATE  9 TO STATE 46
PRODUCTION 371 AND TRANSITION FROM STATE  2 TO STATE  6
TRANSITION FROM STATE  6 TO STATE 18
PRODUCTION 274 AND TRANSITION FROM STATE  6 TO STATE 34
TRANSITION FROM STATE 34 TO STATE 11
PRODUCTION  8 AND TRANSITION FROM STATE 34 TO STATE 113
PRODUCTION 280 AND TRANSITION FROM STATE 34 TO STATE 112
PRODUCTION 440 AND TRANSITION FROM STATE  6 TO STATE 35
PRODUCTION 284 AND TRANSITION FROM STATE 35 TO STATE 116
PRODUCTION 282 AND TRANSITION FROM STATE 116 TO STATE 207
PRODUCTION 275 AND TRANSITION FROM STATE  6 TO STATE 37
PRODUCTION 271 AND TRANSITION FROM STATE  6 TO STATE 38
TRANSITION FROM STATE 38 TO STATE 117
  18 BEGIN
PRODUCTION 276 AND TRANSITION FROM STATE  6 TO STATE 39
PRODUCTION 25 AND TRANSITION FROM STATE 39 TO STATE 119
PRODUCTION 441 AND TRANSITION FROM STATE  6 TO STATE 40
TRANSITION FROM STATE 40 TO STATE 104
  19    NULL;
PRODUCTION 204 AND TRANSITION FROM STATE 104 TO STATE 195
TRANSITION FROM STATE 195 TO STATE 313
PRODUCTION 219 AND TRANSITION FROM STATE 195 TO STATE 338
PRODUCTION 208 AND TRANSITION FROM STATE 195 TO STATE 340
PRODUCTION 202 AND TRANSITION FROM STATE 104 TO STATE 197
TRANSITION FROM STATE 197 TO STATE 341
  20 END MAIN;
PRODUCTION 200 AND TRANSITION FROM STATE 104 TO STATE 198
PRODUCTION 259 AND TRANSITION FROM STATE 198 TO STATE 343
PRODUCTION 258 AND TRANSITION FROM STATE 40 TO STATE 120
TRANSITION FROM STATE 120 TO STATE 209
TRANSITION FROM STATE 209 TO STATE 11
PRODUCTION  8 AND TRANSITION FROM STATE 209 TO STATE 113
PRODUCTION 280 AND TRANSITION FROM STATE 209 TO STATE 351
PRODUCTION 279 AND TRANSITION FROM STATE 209 TO STATE 352
PRODUCTION 277 AND TRANSITION FROM STATE  6 TO STATE 36
PRODUCTION 378 AND TRANSITION FROM STATE  6 TO STATE 20
PRODUCTION 380 AND TRANSITION FROM STATE  6 TO STATE 24
PRODUCTION 372 AND TRANSITION FROM STATE  2 TO STATE  4
TRANSITION FROM STATE  4 TO STATE 13
PRODUCTION  4 AND TRANSITION FROM STATE  2 TO STATE  5
PRODUCTION  3 AND TRANSITION FROM STATE  2 TO STATE 10
TRANSITION FROM STATE 10 TO STATE 48
```

IV. Source listing

INTERPRETER SOURCE LISTING

```
 1  PROCEDURE INTERPRET;
 2    (*INTERPRETATION OF THE RELATIONAL OPERATORS IS DEPENDENT ON THE VALUE OF
 3      ORD (FALSE).  PROPER OPERATION REQUIRES THE ORD OF FALSE TO EQUAL 0 *)
 4    CONST
 5      NUMPROCESSORS = 3;        (*NUMBER OF SYSTEM PROCESSORS*)
 6      PRIMARYPROCESSOR = 1;     (*PROCESSOR WHICH INITIALIZES THE SYSTEM*)
 7      EXECUTIONLENGTH = 5;      (*NUMBER OF INSTRUCTIONS EXECUTED PER
 8                                  TIMESLICE IN THE SIMULATION*)
 9      (*DESCRIPTION OF THE TASK ACTIVATION RECORD*)
10      SLINKO = 0;    DLINKO     =  1;  PCO      =  2; TASKFLAGO  =  3;
11      ANTO   = 4;    WAITO      =  5;  EXCEPTO =  6;  PRIORITYO = 7;
12      TOFF   = 8;     BASEO      =  9;  TBASEO  = 10;  LINKO    = 11;
13      HEAPO = 12;  DATALOCKO = 13;  CALLERO = 14;  RETURNO = 15;  ENTRYO = 16;
14
15      (*DESCRIPTION OF THE ENTRY FRAME*)
16      ENTRYFRAMESIZE = 3;
17      EGATEO = 0;    EADDRO = 1;  EQUEO = 2;
18
19    TYPE
20      PRIORITIES = 0..5;
21      MACHINEDESCRIPTION = RECORD
22      (*REGISTERS*)  PC,              (*PROGRAM COUNTER*)
23                     HEAP,            (*HEAP POINTER*)
24                     BASE,            (*PTR TO ACTIVE STACK FRAME*)
25                     T:ADDRESSRANGE;(*TOP OF STACK POINTER*)
26                     IR : INSTRUCTION; (*INSTRUCTION REGISTER*)
27      (*HOUSEKEEPING*)
28                     STATE :  (BUSY,IDLE);
29                     CURRENTJOB : ADDRESSRANGE;(*PTR TO TASK ACTIVATION RECORD*)
30                     ICOUNT : INTEGER;  (*LAST INSTRUCTION TO EXECUTE*)
31        END (*MACHINE_DESCRIPTION*);
32
33    VAR
34      S : ARRAY [1..MEMORYSIZE] OF INTEGER;
35      READY : ARRAY [PRIORITIES] OF ADDRESSRANGE;
36      PROCESSOR : ARRAY [1..NUMPROCESSORS] OF MACHINEDESCRIPTION;
37      CURRENTPROCESSOR : INTEGER;    (*SIMULATED-ACTIVE PROCESSOR*)
38      NEWPTR,TEMPTR, I, EFRAMEPTR,TEMPBASE : INTEGER;     (*INDEXES*)
```

```
39      TERMINATION : BOOLEAN;          (*TERMINATE SIMULATION FLAG*)
40
41    FUNCTION FINDBASE (LEV, TEMPBASE : INTEGER) : INTEGER;
42      (*FUNCTION FINDBASE RETURNS THE BASE OF THE STACKFRAME 'LEV' STATIC
43        LINKS DOWN THE STACK*)
44    BEGIN
45      WHILE LEV > 0 DO BEGIN
46        TEMPBASE := S [TEMPBASE];
47        LEV := LEV - 1
48        END  (*WHILE*);
49      FINDBASE := TEMPBASE
50    END  (*FINDBASE*);
51
52
53    PROCEDURE ASSIGN  (QNAME : PRIORITIES ; PROCESSORNUM : INTEGER);
54      (*PROCEDURE ASSIGN REMOVES THE GIVEN TASK FROM THE QUEUE WHERE IT WAITS
55        AND INITIALIZES THE GIVEN PROCESSOR WITH THE DATA NECESSARY TO BEGIN
56        EXECUTION.  THE TASK IS SPECIFIED BY AN INTEGER WHICH POINTS TO THE STACK
57        FRAME OF THE TASK.  THE INITIALIZATION DATA IS STORED IN THE TASK'S
58        STACK FRAME.*)
59      VAR  TASKPTR : INTEGER;  (*PTR TO TASK ACTIVATION RECORD*)
60    BEGIN
61      (*UNLINK THE GIVEN TASK FROM ITS QUEUE*)
62      TASKPTR := READY [QNAME];
63      READY [QNAME] := S [TASKPTR + LINKO];
64
65      (*INITIALIZE THE PROCESSOR -- THE REQUIRED DATA IS STORED IN
66        THE STACKFRAME POINTED TO BY 'TASKPTR'*)
67      WITH PROCESSOR [PROCESSORNUM] DO BEGIN
68        PC := S [TASKPTR + PCO];
69        BASE := S [TASKPTR + BASEO];
70        T := S [TASKPTR + TOFF];
71        HEAP := S [TASKPTR + HEAPO];
72        STATE := BUSY;
73        CURRENTJOB := TASKPTR;
74        ICOUNT := 0;
75        END  (*WITH PROCESSORNUM*)
76    END  (*ASSIGN*);
```

```
 77
 78    PROCEDURE SCHEDULE;
 79      (*PROCEDURE SCHEDULE ASSIGNS IDLE PROCESSORS TO WAITING TASKS.
 80        SCHEDULING CONTINUES FOR AS LONG AS THERE ARE PROCESSORS THAT
 81        ARE IDLE AND TASKS THAT ARE WAITING IN THE READY QUEUE*)
 82
 83      FUNCTION WAITINGTASK : PRIORITIES;
 84        (*FUNCTION WAITING TASK SEARCHES THE READY QUEUES IN ORDER OF PRIORITY
 85          (WHERE PRIORITY 5 IS THE HIGHEST AND PRIORITY 1 IS THE LOWEST)
 86          AND RETURNS THE QUEUE NUMBER OF THE HIGHEST-PRIORIY WAITING TASK*)
 87
 88        VAR MARKER : PRIORITIES;
 89      BEGIN
 90        MARKER := 5;  (*BEGIN THE SEARCH WITH THE HIGHEST PRIORITY QUEUE*)
 91        WHILE READY [MARKER] = 0 DO MARKER := MARKER - 1;
 92        WAITINGTASK := MARKER
 93      END  (*WAITING_TASK*);
 94
 95      FUNCTION IDLEPROCESSOR : INTEGER;
 96        (*FUNCTION IDLE PROCESSOR SEARCHES FOR AN IDLE SYSTEM PROCESSOR.
 97          THE FUNCTION RETURNS THE NUMBER OF THE FIRST IDLE PROCESSOR IT FINDS*)
 98
 99      VAR  PINDEX : INTEGER;
100           FOUND : BOOLEAN;
101      BEGIN
102        FOUND := FALSE; PINDEX := 1;
103        WHILE (PINDEX <= NUMPROCESSORS) AND (NOT FOUND) DO
104          IF PROCESSOR [PINDEX].STATE = IDLE THEN
105            FOUND := TRUE
106          ELSE
107            PINDEX := PINDEX + 1;
108        IF FOUND THEN
109          IDLEPROCESSOR := PINDEX
110        ELSE
111          IDLEPROCESSOR := 0;
112      END  (*FUNCTION IDLEPROCESSOR*);
113
114    BEGIN  (*SCHEDULE_TASK*)
```

```
115       WHILE (IDLEPROCESSOR <> 0) AND (WAITINGTASK <> 0) DO
116         ASSIGN (WAITINGTASK, IDLEPROCESSOR)
117     END  (*SCHEDULE*);
118
119     FUNCTION NEWP (VAR HEAPTR : INTEGER; NUMWORDS : INTEGER) : INTEGER;
120     BEGIN
121       NEWP := HEAPTR + 1;
122       HEAPTR := HEAPTR + NUMWORDS
123     END  (*NEW*);
124
125   PROCEDURE ENTERINQ (TFRAME, PRIORITY : INTEGER);
126     (*PROCEDURE ENTERINQ ENTERS TASK 'TFRAME' IN THE SPECIFIED READY QUEUE*)
127   BEGIN
128     S[TFRAME + LINKO] := 0;
129     IF READY [PRIORITY] <> 0 THEN BEGIN (*TASKS IN QUEUE*)
130       TEMPTR := READY [PRIORITY];
131       WHILE S[TEMPTR + LINKO] <> 0 DO
132         TEMPTR := S[TEMPTR + LINKO];
133       (*TEMPTR NOW POINTS TO THE LAST TASK IN THE QUEUE*)
134       S [TEMPTR + LINKO] := TFRAME;
135       END (*TASKS IN QUEUE*)
136     ELSE  (*QUEUE EMPTY*)
137       READY [PRIORITY] := TFRAME;
138   END (*ENTER_IN_QUEUE*);
139
140
141
142   BEGIN  (*INTERPRET*)
143     (*SET STATE OF ALL PROCESSORS TO IDLE*)
144     FOR I := 1 TO NUMPROCESSORS DO PROCESSOR [I].STATE := IDLE;
145
146     (*INITIALIZE PRIMARY PROCESSOR*)
147     WITH PROCESSOR [PRIMARYPROCESSOR] DO BEGIN
148       T := 0;
149       BASE := 1;
150       PC := 0;
151       HEAP := MEMORYSIZE;
152       STATE := BUSY;
```

```
153       ICOUNT := 0;
154       CURRENTJOB := 1;
155       (*INITIALIZE THE STACKFRAME*)
156       FOR I := 1 TO TASKFRAMESIZE  DO
157         S[I] := 0;
158       S[BASE + TASKFLAGO] := 1;   (*FLAG SET*)
159       S[BASE + PRIORITYO] := 5;   (*FIX*)
160       END  (*WITH PRIMARY PROCESSOR*);
161     CURRENTPROCESSOR := PRIMARYPROCESSOR;
162
163
164     (*INITIALIZE READY QUEUE*)
165     FOR I := 1 TO 5 DO READY [I] := 0;
166     READY [0] := 1;
167
168     (*BEGIN SIMULATION - PRIMARY PROCESSOR ACTIVE*)
169     TERMINATION := FALSE;
170     WHILE NOT TERMINATION DO BEGIN
171         WITH PROCESSOR [CURRENTPROCESSOR], IR DO
172          REPEAT
173             IR := CODE [PC];  PC := PC + 1; ICOUNT := ICOUNT + 1;
174             CASE OP OF
175
176   (* I/O OPERATIONS *)
177
178   IGET : BEGIN
179     READ (S[T+1]);
180     S[FINDBASE (LEVEL,BASE) + ADDR] := S[T+1];
181     END  (*IGET*);
182
183   IPUT : BEGIN  (*PRINT VALUE ON TOP OF THE STACK*)
184     IF LEVEL = 0 THEN (*PUT WITHOUT CARRIAGE RETURN*)
185       WRITE (S[T])
186     ELSE
187       WRITELN (S[T]);
188     T := T - 1;
189     END  (*IPUT*);
190
```

```
191  SPUT : BEGIN  (*PRINT STRING -- LENGTH STORED IN ADDR FIELD*)
192    FOR I := 1 TO ADDR DO BEGIN
193      WRITE (CHR (CODE[PC].ADDR):1);
194      PC := PC + 1;
195      END (*FOR*);
196    IF LEVEL = 1 THEN (*PUT WITH CARRIAGE RETURN*)
197      WRITELN;
198    END  (*SPUT*);
199
200  ACCEPT :  BEGIN  (*ACCEPT, 0, ENTRY # *)
201    (*THIS INSTRUCTION WAS WRITTEN WITH THE ASSUMPTION THAT ENTRYFRAMES
202      IMMEDIATELY FOLLOW THE ENTRY OFFSET IN THE STACK FRAME.  I.E.
203      THE FIRST ENTRY FRAME BEGINS AT BASE + ENTRYO + 1*)
204    (*COMPUTE A POINTER TO THE TASK'S 1ST ENTRY FRAME*)
205    EFRAMEPTR := CURRENTJOB + ENTRYO + 1 + (IR.ADDR - 1)*ENTRYFRAMESIZE;
206    (*SEE IF THIS ENTRY HAS ANY TASKS WAITING*)
207    IF S[EFRAMEPTR + EQUEO] = 0 THEN BEGIN (*NO TASKS WAITING*)
208      S[EFRAMEPTR + EGATEO] := 1;          (*OPEN GATE*)
209
210      (*GO TO SLEEP*)
211      S [CURRENTJOB + PCO] := PC - 1;    (*POINTS TO THE ACCEPT INSTRUCTION*)
212      S [CURRENTJOB + HEAPO] := HEAP;
213      S [CURRENTJOB + BASEO] := BASE;
214      S [CURRENTJOB + TOFF] := T;
215      S [CURRENTJOB + TBASEO] := CURRENTJOB;
216      (*RELEASE PROCESSOR*)
217      PROCESSOR [CURRENTPROCESSOR].STATE := IDLE;
218      ICOUNT := ICOUNT + EXECUTIONLENGTH;
219      SCHEDULE;  (*A NEW PROCESSOR IS SELECTED ON EXIT FROM LOOP*)
220      END (*NO TASKS WAITING*)
221    ELSE BEGIN (*WAITING TASK*)
222      (*REMOVE TASK FROM QUEUE*)
223      S[CURRENTJOB + CALLERO] := S[EFRAMEPTR + EQUEO];
224      S[EFRAMEPTR + EQUEO] := S[S[BASE + CALLERO] + LINKO];
225      END (*WAITING TASK*);
226    END (*ACCEPT*);
227
228  ACTIVATE :  BEGIN  (*ACTIVATE, 0, TASKPTR);  (DATA, PRIORITY, HEAP);
```

```
229                       (DATA, 0, PC);  (DATA, #ENTRIES, T*)
230     (*'TASKPTR' IS RELATIVE TO THE BASE OF THE PARENT.  HEAP AND T ARE
231       RELATIVE TO 'TASKPTR'.  THIS INSTRUCTION WAS WRITTEN WITH THE
232       ASSUMPTION THAT ENTRY FRAMES IMMEDIATELY FOLLOW THE ENTRY OFFSET
233       IN THE STACK FRAME.  I.E. THE FIRST ENTRY FRAME BEGINS AT
234       BASE + ENTRO + 1*)
235
236     REPEAT
237       (*INITIALIZE THE STACKFRAME*)
238       (*NOTE : THE ACTIVATE INSTRUCTION REMAINS IN THE IR THROUGHOUT THE
239         INSTRUCTION'S EXECUTION, EVEN THOUGH PC IS INCREMENTED*)
240       (*SET 'TEMPTR' TO THE BASE OF THE TASK BEING ACTIVATED*)
241       TEMPTR := BASE + IR.ADDR;
242       S [TEMPTR + TBASEO] := TEMPTR;  (* RECORD TASK BASE *)
243
244       S [TEMPTR + SLINKO] :=  BASE;  (*STATIC LINK ASSIGNMENT*)
245       S [TEMPTR + DLINKO] := BASE;  (*DYNAMIC LINK ASSIGNMENT*)
246       S [TEMPTR + PRIORITYO] := CODE [PC].LEVEL;  (*PRIORITY*)
247       S [TEMPTR + BASEO] := TEMPTR;        (*BASE*)
248       S [TEMPTR + HEAPO] := TEMPTR + CODE [PC].ADDR;  (*HEAP PTR INITIALIZATION*)
249       PC := PC + 1;
250       S [TEMPTR + PCO] := CODE [PC].ADDR;     (*PROGRAM COUNTER*)
251       PC := PC + 1;
252       S [TEMPTR + TOFF] := TEMPTR + CODE [PC].ADDR;      (*STACK POINTER*)
253       S [TEMPTR + ENTRYO] :=  CODE [PC].LEVEL;  (* # NUMBER OF ENTRIES*)
254
255       (*CLOSE ALL ENTRIES*)
256       FOR I := TEMPTR + ENTRYO + 1 TO TEMPTR + ENTRYO +
257         ENTRYFRAMESIZE*CODE[PC].LEVEL DO
258         S [I] := 0;  (*CLOSED*)
259       PC := PC + 1;  (*PC NOW POINTS PAST THE LAST DATA WORD OF THE INSTRUCTION*)
260       (*INITIALIZE MISCELLANEOUS CONTROL WORDS IN THE STACKFRAME*)
261       S [TEMPTR + ANTO] := 0;  (*ACTIVE NESTED TASKS*)
262       S [TEMPTR + LINKO] := 0; (*LINK*)
263       S [TEMPTR + DATALOCKO] := 0;  (*UNLOCKED*)
264       S [TEMPTR + EXCEPTO] := 0;
265       S [TEMPTR + TASKFLAGO] := 1;
266       S [TEMPTR + WAITO] := 0;
```

```
267       S [TEMPTR + CALLERO] := 0;
268       S [TEMPTR + RETURNO] := 0;
269
270       (*INCREMENT THE ACTIVE NESTED TASK COUNTER OF THE PARENT --
271         NOTE: PARENT EXECUTES THIS INSTRUCTION*)
272       S [BASE + ANTO] := S [BASE + ANTO] + 1;
273
274       ENTERINQ (TEMPTR, S [TEMPTR + PRIORITYO]);
275       IR := CODE [PC];
276       IF IR.OP = ACTIVATE THEN PC := PC + 1;
277     UNTIL IR.OP <> ACTIVATE;
278     (*UNLOCK QUEUES*)
279     SCHEDULE;  (*GO TO SLEEP??*)
280     END  (*ACTIVATE*);
281
282   CALLENTRY : BEGIN  (*CALLENTRY, LEX DIFF, TASKPTR);  (DATA, ENTRY#, #ENTRIES*)
283     (*THIS PROCEDURE WAS WRITTEN WITH THE ASSUMPTION THAT ENTRY FRAMES
284       IMMEDIATELY FOLLOW ENTRYO IN THE STACK FRAME.  I.E. THE FIRST
285       ENTRY FRAME MUST BEGIN AT BASE + ENTRYO + 1*)
286
287     (*'TASKPTR' IS RELATIVE TO THE BASE OF THE PARENT--
288       COMPUTE AN ABSOLUTE ADDRESS FOR THE BASE OF THE CALLED TASK*)
289     TEMPBASE (*TASK'S ABSOLUTE ADDRESS*) := FINDBASE (LEVEL, BASE) + IR.ADDR;
290
291     (*LINK CALLER TO THE CALLED TASK'S ENTRY QUEUE*)  (*DATA LOCK*)
292     (*ZERO CALLER'S LINK FIELD*)
293     S[CURRENTJOB + LINKO] := 0;
294     (*COMPUTE A POINTER TO THE CALLED ENTRY'S QUEUE*)
295     EFRAMEPTR := TEMPBASE + ENTRYO + 1 + (CODE[PC].LEVEL - 1)*ENTRYFRAMESIZE;
296     TEMPTR := EFRAMEPTR + EQUEO;  (*TEMPTR NOW POINTS TO THE HEAD OF
297                                     THE ENTRY QUEUE*)
298     IF S[TEMPTR] = 0 THEN (*QUEUE EMPTY*)
299       S[TEMPTR] := CURRENTJOB   (*LINK CALLER TO THE QUEUE*)
300     ELSE BEGIN (*TASKS IN QUEUE*)
301       (*FIND THE END OF THE QUEUE*)
302       TEMPTR := S[TEMPTR];  (*SET TEMPTR TO THE FIRST TASK IN THE QUEUE*)
303       WHILE S[TEMPTR + LINKO] <> 0 DO
304         TEMPTR := S [TEMPTR + LINKO];
```

```
305       (*TEMPTR POINTS TO THE LAST TASK IN QUEUE*)
306       S [TEMPTR + LINKO] := CURRENTJOB;  (*LINK CALLER TO QUEUE*)
307       END (*TASKS IN QUEUE*);
308     (*CHECK TO SEE IF THE CALLED TASK'S ENTRY IS OPEN*)
309     IF S [EFRAMEPTR + EGATEO] = 1 THEN BEGIN (*ENTRY OPEN*)
310       I := TEMPBASE + ENTRYO + 1;
311       WHILE I <= TEMPBASE + ENTRYO + CODE[PC].ADDR * ENTRYFRAMESIZE DO BEGIN
312         S [I + EGATEO] := 0;  (*CLOSED*)
313         I := I + ENTRYFRAMESIZE;
314         END (*WHILE*);
315       ENTERINQ (TEMPBASE, S[TEMPBASE + PRIORITYO]);
316       END;
317     (*GO TO SLEEP -- SAVE THE MACHINE'S REGISTERS IN CALLER'S TASKFRAME*)
318     S [CURRENTJOB + PCO] := PC + 1;
319     S [CURRENTJOB + HEAPO] := HEAP;
320     S [CURRENTJOB + BASEO] := BASE;
321     S [CURRENTJOB + TOFF] := T;
322
323     (*RELEASE PROCESSOR*)
324     PROCESSOR [CURRENTPROCESSOR].STATE := IDLE;
325     ICOUNT := ICOUNT + EXECUTIONLENGTH;
326
327     SCHEDULE;  (*DATA LOCK*)
328     END  (*CALL ENTRY*);
329
330   RELEASE : BEGIN  (*RELEASE, 0, ENTRY# *)
331     (*RETRIEVE THE POINTER TO THE BASE OF THE CALLING TASK'S TASKFRAME*)
332     TEMPTR := S [CURRENTJOB + CALLERO];
333     ENTERINQ (TEMPTR, S[TEMPTR + PRIORITYO]);
334     SCHEDULE;
335
336     (*GIVE UP PROCESSOR ??  DATA LOCKS*)
337     END (*RELEASE*);
338
339   TERMINATE : BEGIN  (*TERMINATE,0,0*)
340     TEMPTR := FINDBASE (1, BASE);  (*SET TEMPTR TO THE PARENT'S TASK FRAME*)
341     S [TEMPTR + ANTO] := S [TEMPTR + ANTO] - 1;
342     IF (S [TEMPTR + ANTO] = 0) AND
```

```
343         (S [TEMPTR + WAITO] = 1) THEN  (*PARENT WAITING TO TERMINATE*)
344         (*WAKE UP PARENT*)
345         ENTERINQ (TEMPTR, S[TEMPTR + PRIORITYO]);
346
347       (*TERMINATE SELF*)
348       PROCESSOR [CURRENTPROCESSOR].STATE := IDLE;
349       ICOUNT := ICOUNT + EXECUTIONLENGTH;
350       SCHEDULE;
351       END (*TERMINATE*);
352
353     KILLTASK : BEGIN  (*KILLTASK, 0, TASKPTR*)
354       (*** NOTE :  THIS INSTRUCTION HAS NOT BEEN TESTED ***)
355       (*COMPUTE AN ABSOLUTE ADDRESS FOR THE CALLED TASK'S STACKFRAME*)
356       TEMPBASE (*TASK'S ABSOLUTE ADDRESS*) := FINDBASE (LEVEL,BASE) + IR.ADDR;
357       (*FIND THE PROCESSOR EXECUTING THE TASKTO BE KILLED*)
358       TEMPTR := 0;
359       FOR I := 1 TO NUMPROCESSORS DO
360         IF PROCESSOR [I].CURRENTJOB = TEMPBASE THEN
361           TEMPTR := I;
362       IF TEMPTR <> 0 THEN BEGIN (*FOUND*)
363         (*DECREMENT ANT COUNTER OF PARENT*)
364         I := FINDBASE (1,IR.ADDR);
365         S [I + ANTO] := S[I + ANTO] - 1;
366         (*TELL PROCESSOR TO STOP*)
367         PROCESSOR [TEMPTR].STATE := IDLE;
368         PROCESSOR [TEMPTR].CURRENTJOB := 0;
369         SCHEDULE;
370         END (*FOUND*)
371       ELSE
372         (*LOOK IN THE READY QUEUES, AND REMOVE IF FOUND*);
373       END  (*KILL_TASK*);
374
375     ZAND : BEGIN
376         T := T - 1;
377         S[T] := S[T] + S[T + 1];
378         IF S[T] = 2 THEN
379           S[T] := ORD (TRUE)
380         ELSE
```

```
381           S[T] := ORD (FALSE)
382         END  (*XAND*);
383
384   CALL : BEGIN
385         S[T + SLINKO + 1] := FINDBASE (LEVEL,BASE);
386         S[T + DLINKO + 1] := BASE;
387         S[T + PCO + 1] := 0;
388         S[T + 1 + TASKFLAGO] := 0;  (*FALSE*)
389         S[T + 1 + ANTO] := 0;
390         S[T + 1 + WAITO] := 0;
391         S[T + 1 + HEAPO] := 0;
392         S[T + 1 + BASEO] := 0;
393         S[T + 1 + TBASEO] := 0;
394         S[T + 1 + TOFF] := 0;
395         S[T + 1 + LINKO] := 0;
396         S[T + 1 + PRIORITYO] := 5; (*FIX*)
397         S[T + 1 + CALLERO] := 0;
398         S[T + 1 + EXCEPTO] := 0;
399         S[T + 1 + DATALOCKO] := 0;
400         S[T + 1 + ENTRYO] := 0;
401         S[T + 1 + RETURNO] := PC;
402         BASE := T + 1;  PC := ADDR
403           END;
404
405   PARAMSHIFT :  BEGIN  (*PARAMSHIFT, # OF WORDS, T-INCREMENT*)
406     T := T + IR.ADDR;
407     FOR I := 1 TO IR.LEVEL DO
408       S[T - (I-1)] := S[(T-IR.ADDR) - (I-1)];
409     END  (*PARAMSHIFT*);
410
411   CONCAT :      ;
412
413   EQUAL : BEGIN  T := T - 1;  S[T] := ORD (S[T] = S[T + 1])
414           END;
415
416   (*EXPON : BEGIN  T := T - 1;  S[T] := S[T] ** S[T + 1]
417         END;  MODIFY FOR DIFFERENT ARGUMENT TYPES*)
418
```

```
419  GTR : BEGIN  T := T - 1 ;  S[T] := ORD (S[T] > S[T + 1])
420        END;
421
422  GTREQ : BEGIN  T := T - 1;  S[T] := ORD (S[T] >= S[T + 1])
423          END;
424
425  IADD : BEGIN  T := T - 1;  S[T] := S[T] + S[T + 1]
426         END;
427
428  IDIV : BEGIN  T := T - 1;  S[T] := S[T] DIV S[T + 1]
429       END;
430
431  IMULT : BEGIN  T := T - 1;  S[T] := S[T] * S[T + 1]
432        END;
433
434  INEGATE :   S[T] := -S[T];
435
436  ISUB : BEGIN  T := T - 1;  S[T] := S[T] - S[T + 1]
437       END;
438
439  XIN : BEGIN
440      T := T - 2;
441      IF (S[T] >= S[T + 1]) AND (S[T] <=S[T + 2]) THEN
442        S[T] := ORD (TRUE)
443      ELSE
444        S[T] := ORD (FALSE)
445      END  (*XIN*);
446
447  INCT : T := T + ADDR;
448
449  JMP : PC := ADDR;
450
451  JMPF : BEGIN
452         IF S [T] = 0 THEN PC := ADDR;
453         T := T - 1
454         END;
455
456  JMPT : BEGIN
```

```
457           IF S [T] = 0 THEN PC := ADDR;
458           T := T - 1
459           END;
460
461   LESS : BEGIN  T := T - 1;  S[T] := ORD (S[T] < S[T + 1])
462          END;
463
464   LESSEQ : BEGIN  T := T - 1;  S[T] := ORD (S[T] <= S[T + 1])
465           END;
466
467   ILOAD : BEGIN  T := T + 1;  S[T] := S [FINDBASE (LEVEL,BASE) + ADDR]
468           END;
469
470   ENTILOAD :  BEGIN
471     (*SET TEMPTR TO THE BASE OF CALLER'S TASKFRAME*)
472     TEMPTR := S [CURRENTJOB + CALLERO];
473     (*SET TEMPTR TO THE TOP OF THE CALLER'S STACK*)
474     TEMPTR := S [TEMPTR + TOFF];
475     T := T + 1;
476     S[T] := S[TEMPTR + IR.ADDR];
477     END  (*ENTILOAD*);
478
479   ENTISTORE : BEGIN
480     (*SET TEMPTR TO THE BASE OF THE CALLER'S TASKFRAME*)
481     TEMPTR := S[CURRENTJOB + CALLERO];
482     (*SET TEMPTR TO THE TOP OF THE CALLER'S STACK*)
483     TEMPTR := S[TEMPTR + TOFF];
484     S[TEMPTR + IR.ADDR] := S[T];
485     T := T - 1;
486     END (*ENTISTORE*);
487
488   ILOADCONST : BEGIN T := T + 1;  S[T] := ADDR
489           END;
490
491   RLOADCONST : (*DEFINE REALS*);
492
493   IMOD : BEGIN  T := T - 1;  S[T] := S[T] MOD S[T + 1]
494         END;
```

```
495
496
497   NOTEQ : BEGIN  T := T - 1;  S[T] := ORD ( S[T] <> S[T] + 1)
498           END;
499   RADD : (*DEFINE FLOATING POINT FORMAT*);
500
501   RAISE : (*FILL IN THE BLANK*) ;
502
503   RDIV : (*DEFINE FLOATING POINT FORMAT*);
504
505   REM : (*WRITE ROUTINE*);
506
507   RETURN : BEGIN
508     (*SEE IF THERE ARE ANY ACTIVE NESTED TASKS*)
509     IF S [BASE + ANTO] = 0 THEN BEGIN  (*NONE ACTIVE*)
510       T := BASE - 1;
511       PC := S [BASE + RETURNO];
512       BASE := S [BASE + DLINKO];
513       END (*NONE ACTIVE*)
514     ELSE BEGIN  (*WAIT FOR TASK(S) TO COMPLETE*)
515       (*SET PARENT WAITING FLAG*)
516       S [BASE + WAITO] := 1;
517       (*SET PC BACK TO THE RETURN INSTRUCTION*)
518       S [BASE + PCO] := PC - 1;
519       S [BASE + HEAPO] := HEAP;
520       S [BASE + BASEO] := BASE;
521       S [BASE + TOFF] := T;
522       (*GO TO SLEEP*)
523       PROCESSOR [CURRENTPROCESSOR].STATE := IDLE;
524       ICOUNT := ICOUNT + EXECUTIONLENGTH;
525       SCHEDULE;
526       END (*WAIT*)
527     END (*RETURN*);
528
529   RMULT : (*DEFINE FLOATING POINT FORMAT
530
531   RNEG  :              ""
532
```

```
533  KSUB  :              ""              *);
534
535
536  ISTORE : BEGIN
537         S [FINDBASE(LEVEL,BASE) + ADDR] := S[T];
538         IF SWITCH [TRACESTORE] THEN
539           WRITELN (S[T]);
540         T := T - 1;
541         END;
542
543  ZNEW : BEGIN T := T + 1;  S[T] := NEWP (HEAP,ADDR)
544       END;
545
546  ZNOT : BEGIN
547       IF S[T] = 0 THEN
548         S[T] := ORD (TRUE)
549       ELSE
550         S[T] := ORD (FALSE)
551       END  (*ZNOT*);
552
553  ZOR :  BEGIN
554      T := T - 1;
555      S[T] := S[T] + S[T + 1];
556      IF S[T] = 0 THEN
557        S[T] := ORD (FALSE)
558      ELSE
559        S[T] := ORD (TRUE)
560      END  (*ZOR*);
561
562  ZXOR : BEGIN
563       T := T - 1;
564       S[T] := S[T] + S[T + 1];
565       IF S[T] = 1 THEN
566         S[T] := ORD(TRUE)
567       ELSE
568         S[T] := ORD (FALSE)
569       END  (*ZXOR*);
570
```

```
571             END  (*CASE*)
572           UNTIL (PC = 0) OR (ICOUNT >= EXECUTIONLENGTH);
573           IF PROCESSOR [CURRENTPROCESSOR].PC = 0 THEN (*EXECUTION COMPLETE*)
574             TERMINATION := TRUE
575           ELSE BEGIN (*SELECT THE NEXT PROCESSOR*)
576             CURRENTPROCESSOR := CURRENTPROCESSOR MOD NUMPROCESSORS + 1;
577             I := 0;
578             WHILE (PROCESSOR [CURRENTPROCESSOR].STATE = IDLE) AND
579                   (I <= NUMPROCESSORS) DO BEGIN
580               CURRENTPROCESSOR := CURRENTPROCESSOR MOD NUMPROCESSORS + 1;
581               I := I + 1;
582               END (*WHILE*);
583             IF I > NUMPROCESSORS THEN BEGIN (*ERROR ABORT*)
584               TERMINATION := TRUE;
585               WRITELN (' LOOPING IN INTERPRET -- ALL PROCESSORS IDLE');
586               END (*ERROR ABORT*)
587             ELSE
588               PROCESSOR [CURRENTPROCESSOR].ICOUNT := 0;
589             END  (*SELECT*)
590       END  (*WHILE NOT TERMINATION*)
591   END  (*INTERPRET*);
```

VITA

Alan R. Garlington was born on 20 August 1951 in Chicopee, Massachusetts to Arthur R. Garlington, Jr. and Claire Y. (Cournoyer) Garlington. He attended high school at Rome Free Academy in Rome, New York and graduated in 1969. In June of that year, he entered the USAF Academy in Colorado Springs, Colorado and subsequently graduated with a Bachelor of Science Degree in Electrical Engineering in June of 1973. His first Air Force active duty assignment was to Undergraduate Navigation Training at Mather AFB in Sacramento California. He graduated in April of 1974 and entered Electronic Warfare Training, also at Mather. Upon graduation from Electronic Warfare Training in October of 1974, he was assigned to the 62nd Bombardment Squadron, 2nd Bomb Wing (SAC), at Barksdale AFB, Louisiana. In March 1978 he was assigned duties as wing electronic warfare officer where he served until entering the Air Force Institute of Technology School of Engineering at Wright Patterson AFB, Ohio in Sept 1979. He is a member of Tau Beta Pi and Eta Kappa Nu.

Capt Garlington was married on 29 December 1973 in Rome, New York to Pasqualina J. DiMarco. They have one son, Christopher Alan, who was born on 12 September 1979.

Permanent address: 6623 Williams Road
Rome, New York 13440

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFIT/GCS/MA/81M-1 ✓	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) PRELIMINARY DESIGN AND IMPLEMENTATION OF AN ADA PSEUDO-MACHINE		5. TYPE OF REPORT & PERIOD COVERED MS THESIS 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Alan R. Garlington CAPT, USAF		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Department of Mathematics AFIT/EN		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS		12. REPORT DATE March 81 13. NUMBER OF PAGES 118
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) 15a. DECLASSIFICATION/ DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

21 MAY 1981

18. SUPPLEMENTARY NOTES

APPROVED FOR PUBLIC RELEASE AFR 190-17
Air Force Institute of Technology (ATC)
Wright-Patterson AFB, OH 45433

Fredric C. Lynch
FREDRIC C. LYNCH, Major, USAF
Director of Public Affairs

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Ada
Compilers
Computer Programs
Microcomputers
Minicomputers
Pseudo-machine

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This project involved defining an Ada pseudo-machine and developing an Ada to pseudo-code test translator. The translator's front end incorporates a table-driven parser that can parse the entire proposed-standard Ada language. The translator's semantic routines allow integer data objects, several control structures, procedures, functions, packages and tasks. These routines generate pseudo-code that is executed by an interpreter program included in the translator. The interpreter constitutes a complete description of the pseud-

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 55 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

BLOCK 20 continued:

machine whose architecture consists of multiple, stack-oriented processors that access a common memory. Interesting features of the project include the hash-coded symbol table that supports Ada's visibility rules and the pseudo-machine architecture that supports Ada's tasking.


DAT
FILM
7